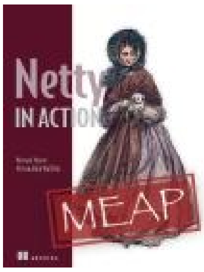
目錄

Essential Netty in Action 《Netty 实战(精髓)》



It is a book about the Essentials of Norman Maurer's Netty in Action(base on MEAP v10). Through this book, you can quickly start with Netty. This is a GitBook version of the book: http://waylau.gitbooks.io/essential-netty-in-action/ Let's READ!

《Netty 实战(精髓)》是对 Norman Maurer 的 《Netty in Action》(基于 MEAP v10)的一个中文精简。取其精华，去其糟粕，带你快速掌握 Netty ，插入配图，图文并茂方便用户理解。本书利用业余时间编写,由于时间紧凑,精力和能力有限,书中未免有纰漏和错误,望读者能够热忱斧正。

对于初学者，也推荐参阅《Netty 4.x 用户指南》。与之类似的 NIO 框架还有 MINA, 可参阅《Apache MINA 2 用户指南》

关于开源

本项目是针对当前市面上缺乏 Netty 的相关的参考资料而产生的，广大 Netty 爱好者通过开源方式来学习交流 Netty，推动 Netty 社区的繁荣。

当然，广大开发者需认识到，开源不等于免费，对于原著的版权，仍应抱有敬意。请支持原著！

如何开始阅读

选择下面入口之一：

- https://github.com/waylau/essential-netty-in-action/ 的 SUMMARY.md（源码）
- http://waylau.gitbooks.io/essential-netty-in-action/ 点击 Read 按钮（同步更新，国内访问速度一般）
- https://waylau.com/essential-netty-in-action/（国内访问速度快，定期更新。最后更新于 2016-2-16）

意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/essential-netty-in-action/issues

联系作者:

您也可以直接联系我：

- 博客：https://waylau.com
- 邮箱：waylau521(at)gmail.com
- 微博：http://weibo.com/waylau521
- 开源：https://github.com/waylau

其他书籍

若您对本书不感冒，笔者还写了其他方面的超过一打的书籍（可见 https://waylau.com/books/），多是开源电子书。

本人也维护了一个books-collection项目，里面提供了优质的专门给程序员的开源、免费图书集合。

开源捐赠

开源不易，熬夜辛苦
请老卫喝一杯烈酒~

支付宝 ALIPAY

捐赠所得所有款项将用于开源事业！

Netty-异步和数据驱动

什么是 Netty

Netty 是一个利用 Java 的高级网络的能力，隐藏其背后的复杂性而提供一个易于使用的 API 的客户端/服务器框架。Netty 提供高性能和可扩展性，让你可以自由地专注于你真正感兴趣的东西，你的独特的应用！

在这一章我们将解释 Netty 在处理一些高并发的网络问题体现的价值。然后，我们将介绍基本概念和构成 Netty 的工具包，我们将在这本书的其余部分深入研究。

一些历史

在网络发展初期，需要花很多时间来学习 socket 的复杂，寻址等等，在 C socket 库上进行编码，并需要在不同的操作系统上做不同的处理。

Java 早期版本(1995-2002)介绍了足够的面向对象的糖衣来隐藏一些复杂性，但实现复杂的客户端-服务器协议仍然需要大量的样板代码（和进行大量的监视才能确保他们是对的）。

这些早期的 Java API（java.net）只能通过原生的 socket 库来支持所谓的“blocking（阻塞）”的功能。一个简单的例子

Listing 1.1 Blocking I/O Example

```
ServerSocket serverSocket = new ServerSocket(portNumber);//1
Socket clientSocket = serverSocket.accept();             //2
BufferedReader in = new BufferedReader(                    //3
        new InputStreamReader(clientSocket.getInputStream()));
PrintWriter out =
        new PrintWriter(clientSocket.getOutputStream(), true);
String request, response;
while ((request = in.readLine()) != null) {                //4
    if ("Done".equals(request)) {                        //5
        break;
    }
}
response = processRequest(request);                        //6
out.println(response);                                   //7
}                                                         //8
```

1.ServerSocket 创建并监听端口的连接请求

2.accept() 调用阻塞，直到一个连接被建立了。返回一个新的 Socket 用来处理 客户端和服务端的交互

3.流被创建用于处理 socket 的输入和输出数据。BufferedReader 读取从字符输入流里面的本文。PrintWriter 打印格式化展示的对象读到本文输出流

4.处理循环开始 readLine() 阻塞，读取字符串直到最后是换行或者输入终止。

5.如果客户端发送的是“Done”处理循环退出

6.执行方法处理请求，返回服务器的响应

7.响应发回客户端

8.处理循环继续

显然，这段代码限制每次只能处理一个连接。为了实现多个并行的客户端我们需要分配一个新的 Thread 给每个新的客户端 Socket(当然需要更多的代码)。但考虑使用这种方法来支持大量的同步，长连接。在任何时间点多线程可能处于休眠状态，等待输入或输出数据。这很容易使得资源的大量浪费，对性能产生负面影响。当然，有一种替代方案。

除了示例中所示阻塞调用，原生 socket 库同时也包含了非阻塞 I/O 的功能。这使我们能够确定任何一个 socket 中是否有数据准备读或写。我们还可以设置标志，因为读/写调用如果没有数据立即返回；就是说，如果一个阻塞被调用后就会一直阻塞，直到处理完成。通过这种方法，会带来更大的代码的复杂性成本，其实我们可以获得更多的控制权来如何利用网络资源。

JAVA NIO

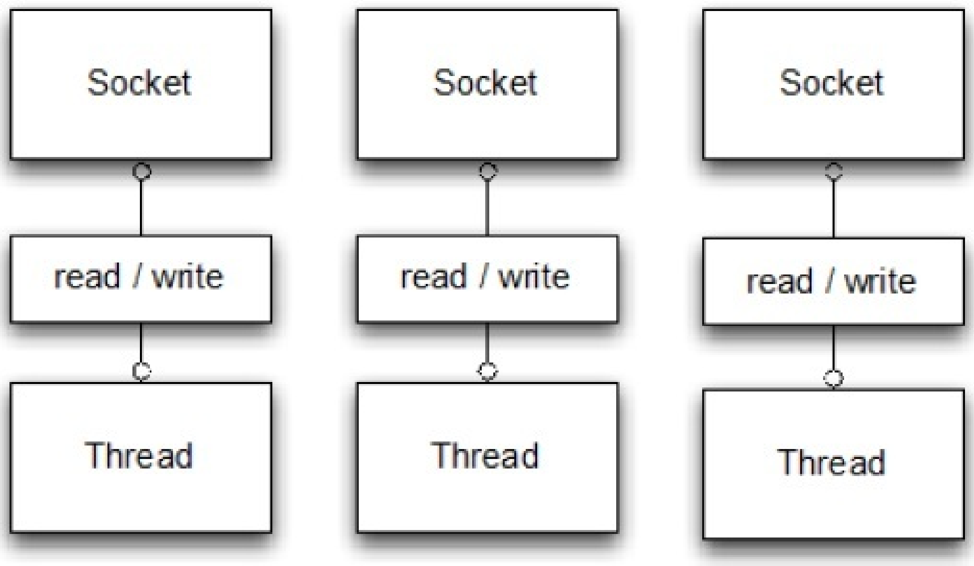
在 2002 年，Java 1.4 引入了非阻塞 API 在 java.nio 包（NIO）。

"New"还是"Nonblocking"?

NIO 最初是为 New Input/Output 的缩写。然而，Java 的 API 已经存在足够长的时间，它不再是新的。现在普遍使用的缩写来表示Nonblocking I/O (非阻塞 I/O)。另一方面，一般（包括作者）指阻塞 I/O 为 OIO 或 Old Input/Output。你也可能会遇到普通 I/O。

我们已经展示了在 Java 的 I/O 阻塞一例例子。图 1.1 展示了方法 必须扩大到处理多个连接：给每个连接创建一个线程，有些连接是空闲的！显然，这种方法的可扩展性将是受限于可以在 JVM 中创建的线程数。

Figure 1.1 Blocking I/O

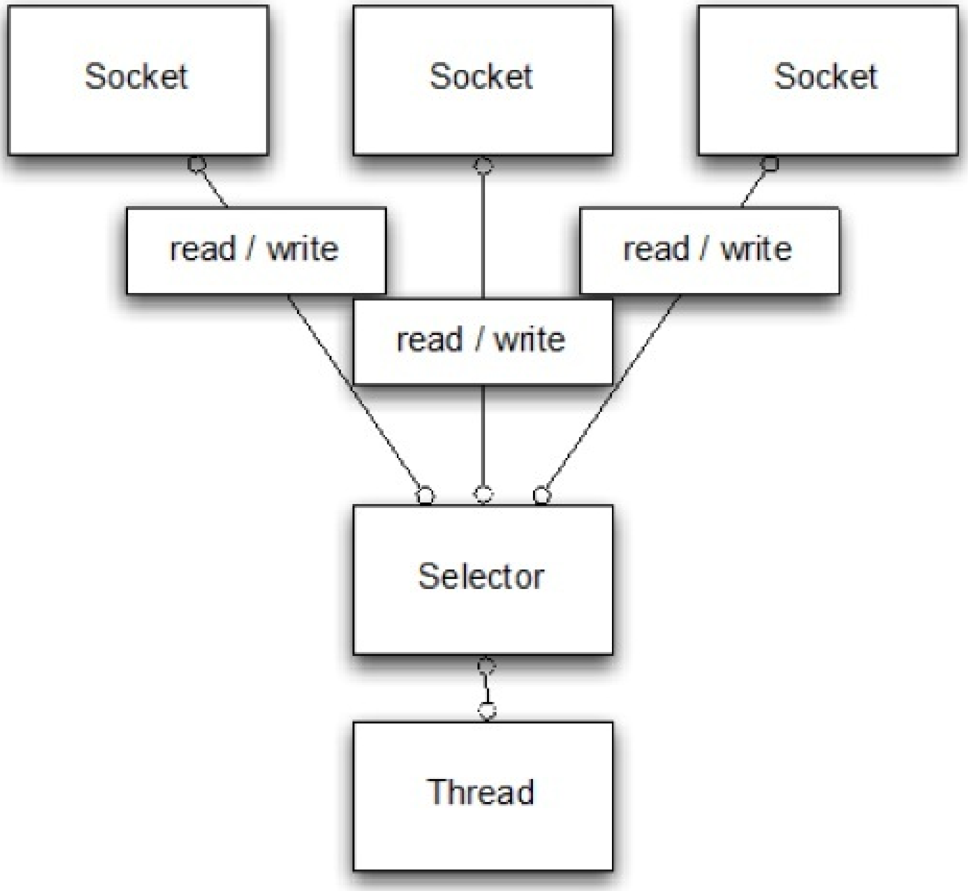


当你的应用中连接数比较少，这个方案还是可以接受。当并发连接超过10000 时，context-switching（上下文切换）开销将是明显的。此外，每个线程都有一个默认的堆栈内存分配了 128K 和 1M 之间的空间。考虑到整体的内存和操作系统需要处理 100000 个或更多的并发连接资源，这似乎是一个不理想的解决方案。

SELECTOR

相比之下，图1.2 显示了使用非阻塞I/O，主要是消除了这些方法 约束。在这里，我们介绍了“Selector”，这是 Java 的无阻塞 I/O 实现的关键。

Figure 1.2 Nonblocking I/O



Selector 最终决定哪一组注册的 socket 准备执行 I/O。正如我们之前所解释的那样，这 I/O 操作设置为非阻塞模式。通过通知，一个线程可以同时处理多个并发连接。（一个 Selector 由一个线程通常处理，但具体实施可以使用多个线程。）因此，每次读或写操作执行能立即检查完成。总体而言，该模型提供了比 阻塞 I/O 模型 更好的资源使用，因为

- 可以用较少的线程处理更多连接，这意味着更少的开销在内存和上下文切换上
- 当没有 I/O 处理时，线程可以被重定向到其他任务上。

你可以直接用这些 Java API 构建的 NIO 建立你的应用程序，但这样做 正确和安全是无法保证的。实现可靠和可扩展的 event-processing（事件处理器）来处理和调度数据并保证尽可能有效地，这是一个繁琐和容易出错的任务，最好留给专家 - Netty。

Netty 介绍

一个应用想要支持成千上万并发的客户端，在以前，这样的想法会被认为是荒谬。而在今天，我们认为这是理所当然的。事实上，开发者知道，总是会有这样的需求——以较低的成本交付来换取更大的吞吐量和可用性。

我们不要低估最后一点的重要性。我们从漫长的痛苦的经验学习到，低级别的 API 不仅暴露了高级别直接使用的复杂性，而且引入了过分依赖于这项技术所造成的短板。因此，面向对象的一个基本原则：通过抽象来隐藏背后的复杂性。

这一原则已见成效，框架的形式封装解决方案成为了常见的编程任务，他们中有许多典型的分布式系统。现在大多数专业的 Java 开发人员都熟悉一个或多个这些框架（比如 Spring），并且许多已成为不可或缺的，使他们能够满足他们的技术要求以及他们的计划。

谁在用 Netty

Netty 是一个广泛使用的 Java 网络编程框架（Netty 在 2011 年获得了Duke's Choice Award，见https://www.java.net/dukeschoice/2011）。它活跃和成长于用户社区，像大型公司 Facebook 和 Instagram 以及流行 开源项目如 Infinispan, HornetQ, Vert.x, Apache Cassandra 和 Elasticsearch 等，都利用其强大的对于网络抽象的核心代码。

反过来，Netty 也从这些开源项目中获益。随着这些项目的作用，Netty 也不断提高了其应用的范围和灵活性，比如已经实现了的协议就有 FTP, SMTP, HTTP, WebSocket 和 SPDY 以及其他二进制和基于文本的协议。

在初创公司中 Firebase 和 Urban Airship 在使用 Netty。前者 Firebase 是使用 long-lived HTTP 连接，后者是使用 各种推送通知。

当你使用 Twitter,你会使用 Finagle,这个是基于 Netty API 提供给内部系统通讯。Facebook 使用 Netty 来提供于 Nifty 类似的功能 Apache Thrift 服务。这些公司可扩展性和高性能的表现得益于 Netty 的贡献。

这些例子的真实案例会在后面几章讲到。

2011 年 Netty 项目从 Red Hat 独立开来从而让广泛的开发者社区贡献者参与进来。Red Hat，Twitter 继续使用 Netty ,并且成为保持其最活跃的贡献者之一。

下面展示了 Netty 技术和方法的特点

- 设计
    - 针对多种传输类型的统一接口 - 阻塞和非阻塞
    - 简单但更强大的线程模型

- 眞正的无连接的数据报套接字支持
- 链接逻辑支持复用
* 易用性
    - 大量的 Javadoc 和 代码实例
    - 除了在 JDK 1.6 + 额外的限制。（一些特征是只支持在Java 1.7 +。可选的功能可能有额外的限制。）
* 性能
    - 比核心 Java API 更好的吞吐量，较低的延时
    - 资源消耗更少，这个得益于共享池和重用
    - 减少内存拷贝
* 健壮性
    - 消除由于慢，快，或重载连接产生的 OutOfMemoryError
    - 消除经常发现在 NIO 在高速网络中的应用中的不公平的读/写比
* 安全
    - 完整的 SSL / TLS 和 StartTLS 的支持
    - 运行在受限的环境例如 Applet 或 OSGI
* 社区
    - 发布的更早和更频繁
    - 社区驱动

异步和事件驱动

所有的网络应用程序需要被设计为可扩展性，可以被界定为“一个系统，网络能力，或过程中能够处理越来越多的工作方式或可扩大到容纳增长的能力”（见 Bondi, André B. (2000). "Characteristics of scalability and their impact on performance"）。我们已经说过，Netty 帮助您利用非阻塞 I/O 完成这一目标，通常称为“异步 I/O”

我们将使用“异步”和其同源词在这本书中大量的使用，所以这是介绍他们的一个很好的时候。异步，即非同步事件，当然是跟你日常生活的类似。例如，您可以发送电子邮件；可能得到或者得不到任何回应，或者当你发送一个您可能会收到一个消息。异步事件也可以有一个有序的关系。例如，你通常不会收到一个问题的答案直到提出一个问题，但是你并没有阻止同时一些其他的东西。

在日常生活中异步就这样发生了，所以我们不会经常想到。但让计算机程序的工作方式，来实现我们提出了的特殊的问题，会有一点复杂。在本质上，一个系统是异步和“事件驱动”将会表现出一个特定的，对我们来说，有价值的 行为：它可以响应在任何时间以任何顺序发生的事件。

这是我们要建立一种制度，正如我们将会看到，这是典范的 Netty 自底向上的支持。

构成部分

正如我们前面解释的，非阻塞 I/O 不会强迫我们等待操作的完成。在这种能力的基础上，真正的异步 I/O 起到了更进一步的作用:一个异步方法完成时立即返回并直接或稍后通知用户。

正如我们将看到的,在一个网络环境的异步模型可以更有效地利用资源,可以快速连续执行多个调用。

Channel

Channel 是 NIO 基本的结构。它代表了一个用于连接到实体如硬件设备、文件、网络套接字或程序组件,能够执行一个或多个不同的 I/O 操作（例如读或写）的开放连接。

现在,把 Channel 想象成一个可以“打开”或“关闭”,“连接”或“断开”和作为传入和传出数据的运输工具。

Callback (回调)

callback (回调)是一个简单的方法,提供给另一种方法作为引用,这样后者就可以在某个合适的时间调用前者。这种技术被广泛使用在各种编程的情况下,最常见的方法之一通知给其他人操作已完成。

Netty 内部使用回调处理事件时。一旦这样的回调被触发，事件可以由接口 ChannelHandler 的实现来处理。如下面的代码，一旦一个新的连接建立了,调用 channelActive(),并将打印一条消息。

Listing 1.2 ChannelHandler triggered by a callback

```
public class ConnectHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {   //1
        System.out.println(
                "Client " + ctx.channel().remoteAddress() + " connected");
    }
}
```

1.当建立一个新的连接时调用 ChannelActive()

Future

Future 提供了另外一种通知应用操作已经完成的方式。这个对象作为一个异步操作结果的占位符,它将在将来的某个时候完成并提供结果。

JDK 附带接口 java.util.concurrent.Future ,但所提供的实现只允许您手动检查操作是否完成或阻塞了。这是很麻烦的，所以 Netty 提供自己了的实现,ChannelFuture,用于在执行异步操作时使用。

ChannelFuture 提供多个附件方法来允许一个或者多个 ChannelFutureListener 实例。这个回调方法 operationComplete() 会在操作完成时调用。事件监听者能够确认这个操作是否成功或者是错误。如果是后者,我们可以检索到产生的 Throwable。简而言之, ChannelFutureListener 提供的通知机制不需要手动检查操作是否完成的。

每个 Netty 的 outbound I/O 操作都会返回一个 ChannelFuture;这样就不会阻塞。这就是 Netty 所谓的“自底向上的异步和事件驱动”。

下面例子简单的演示了作为 I/O 操作的一部分 ChannelFuture 的返回。当调用 connect() 将会直接是非阻塞的，并且调用在背后完成。由于线程是非阻塞的，所以无需等待操作完成，而可以去干其他事，因此这令资源利用更高效。

Listing 1.3 Callback in action

```
Channel channel = ...;
//不会阻塞
ChannelFuture future = channel.connect(
    new InetSocketAddress("192.168.0.1", 25));
```

1.异步连接到远程地址

下面代码描述了如何利用 ChannelFutureListener 。首先，连接到远程地址。接着，通过 ChannelFuture 调用 connect() 来 注册一个新ChannelFutureListener。当监听器被通知连接完成，我们检查状态。如果是成功，就写数据到 Channel，否则我们检索 ChannelFuture 中的Throwable。

注意，错误的处理取决于你的项目。当然,特定的错误是需要加以约束 的。例如,在连接失败的情况下你可以尝试连接到另一个。

Listing 1.4 Callback in action

```
Channel channel = ...;
//不会阻塞
ChannelFuture future = channel.connect(            //1
        new InetSocketAddress("192.168.0.1", 25));
future.addListener(new ChannelFutureListener() {  //2
@Override
public void operationComplete(ChannelFuture future) {
    if (future.isSuccess()) {                    //3
        ByteBuf buffer = Unpooled.copiedBuffer(
                "Hello", Charset.defaultCharset()); //4
        ChannelFuture wf = future.channel().writeAndFlush(buffer);                //5
        // ...
    } else {
        Throwable cause = future.cause();        //6
        cause.printStackTrace();
    }
}
});
```

1.异步连接到远程对等节点。调用立即返回并提供 ChannelFuture。

2.操作完成后通知注册一个 ChannelFutureListener 。

3.当 operationComplete() 调用时检查操作的状态。

4.如果成功就创建一个 ByteBuf 来保存数据。

5.异步发送数据到远程。再次返回ChannelFuture。

6.如果有一个错误则抛出 Throwable,描述错误原因。

Event 和 Handler

Netty 使用不同的事件来通知我们更改的状态或操作的状态。这使我们能够根据发生的事件触发适当的行为。

这些行为可能包括：

- 日志
- 数据转换
- 流控制
- 应用程序逻辑

由于 Netty 是一个网络框架,事件很清晰的跟入站或出站数据流相关。因为一些事件可能触发传入的数据或状态的变化包括:

- 活动或非活动连接
- 数据的读取

- 用户事件
- 错误

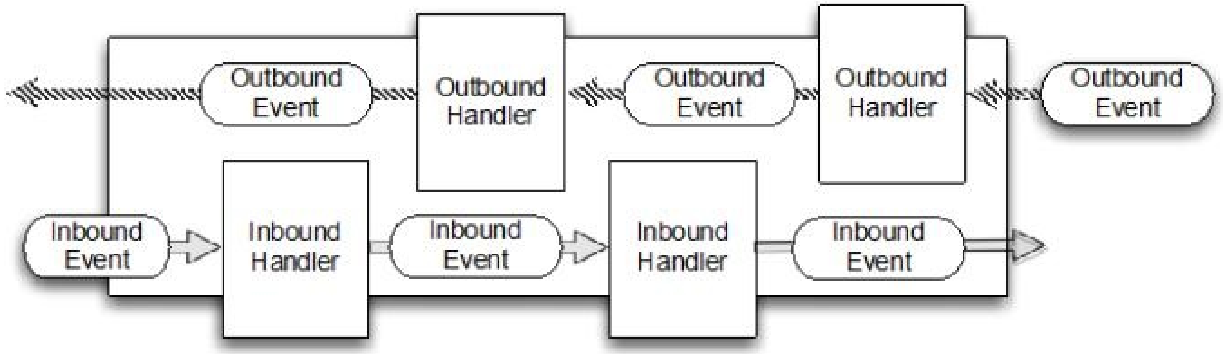
出站事件是由于在未来操作将触发一个动作。这些包括:

- 打开或关闭一个连接到远程
- 写或冲刷数据到 socket

每个事件都可以分配给用户实现处理程序类的方法。这说明了事件驱动的范例可直接转换为应用程序构建块。

图1.3显示了一个事件可以由一连串的事件处理器来处理

Figure 1.3 Event Flow



Netty 的 ChannelHandler 是各种处理程序的基本抽象。想象下，每个处理器实例就是一个回调，用于执行对各种事件的响应。

在此基础之上，Netty 也提供了一组丰富的预定义的处理程序,您可以开箱即用。比如，各种协议的编解码器包括 HTTP 和 SSL/TLS。在内部,ChannelHandler 使用事件和 future 本身,创建具有 Netty 特性抽象的消费者。

整合

FUTURE, CALLBACK 和 HANDLER

Netty 的异步编程模型是建立在 future 和 callback 的概念上的。所有这些元素的协同为自己的设计提供了强大的力量。

拦截操作和转换入站或出站数据只需要您提供回调或利用 future 操作返回的。这使得链操作简单、高效,促进编写可重用的、通用的代码。一个 Netty 的设计的主要目标是促进“关注点分离”:你的业务逻辑从网络基础设施应用程序中分离。

SELECTOR, EVENT 和 EVENT LOOP

Netty 通过触发事件从应用程序中抽象出 Selector，从而避免手写调度代码。EventLoop 分配给每个 Channel 来处理所有的事件，包括

- 注册感兴趣的事件
- 调度事件到 ChannelHandler
- 安排进一步行动

该 EventLoop 本身是由只有一个线程驱动，它给一个 Channel 处理所有的 I/O 事件，并且在 EventLoop 的生命周期内不会改变。这个简单而强大的线程模型消除你可能对你的 ChannelHandler 同步的任何关注，这样你就可以专注于提供正确的回调逻辑来执行。该 API 是简单和紧凑。

关于本书

我们开始通过讨论阻塞和非阻塞处理之间的差异来了解到后一种方法的优点。然后，我们转移到了的 Netty的功能，设计和效益的概述。这些包括了Netty 的异步模型，包括回调，future 及其组合使用。我们还谈到了Netty 的线程模型，事件是如何被使用的，以及它们如何被拦截和处理。展望未来，我们将更加深入探索如何使用这些丰富的工具集用来满足特殊需求的应用。

一路上，我们将介绍公司的工程师自己的案例研究解释为什么他们选择的Netty 以及他们如何使用它。

因此，让我们开始吧。在下一章中，我们将深入研究了 Netty 的 API 的基础知识，编程模型，开始写 echo（回声）服务器和客户端。

第一个 Netty 应用

在本章中，首先你要确保你有一个可以工作的开发环境，并通过构建一个简单的客户端和服务器来进行测试。虽然在开始下一章节前，我们还不会开始学习的 Netty 框架的细节，但在这里我们将会仔细观察我们所引入的 API 方面的内容，即通过 ChannelHandler 来实现应用的逻辑。

设置开发环境

如果你已经有了 Maven 的开发环境，那你可以跳过本节。

本书例子需要 JDK 和 Apache Maven,都可以免费下载到。

1.安装配置 JDK

建议用 JDK 7+

2.下载 IDE

JAVA 的 IDE 很多，主流的有

- Eclipse: http://www.eclipse.org
- NetBeans: http://www.netbeans.org
- Intellij Idea Community Edition: http://www.jetbrains.com

3.下载安装 Maven

可以参考：http://www.waylau.com/apache-maven-3-1-0-installation-deployment-and-use/

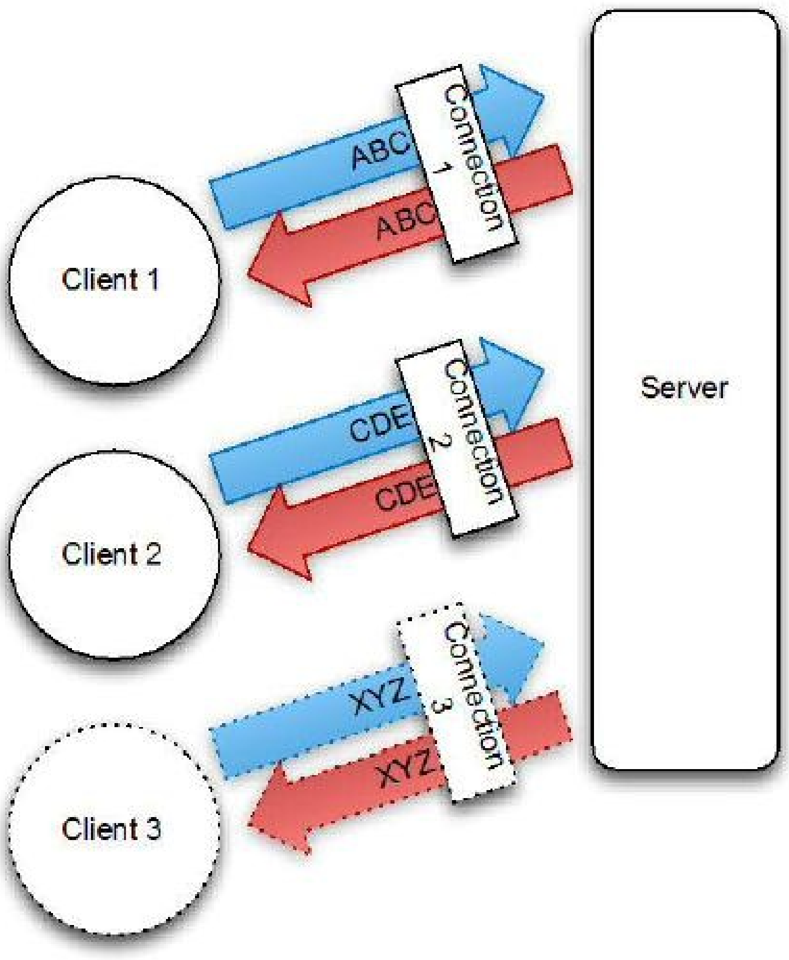
4.配置工具

确保系统环境变量有 JAVA_HOME 和 M2_HOME

Netty 客户端/服务器 总览

在本节中，我们将构建一个完整的的 Netty客 户端和服务器。虽然你可能集中在写客户端是浏览器的基于 Web 的服务，接下来你将会获得更完整了解 Netty 的 API 是如何实现客户端和服务器的。

Figure 2.1.Echo client / server



图中显示了连接到服务器的多个并发的客户端。在理论上，客户端可以支持的连接数只受限于使用的 JDK 版本中的制约。

echo（回声）客户端和服务器之间的交互是很简单的;客户端启动后，建立一个连接发送一个或多个消息发送到服务器，其中每相呼应消息返回给客户端。诚然，这个应用程序并不是非常有用。但这项工作是为了更好的理解请求 - 响应交互本身，这是一个基本的模式的客户端/服务器系统。

我们将通过检查服务器端代码开始。

写一个 echo 服务器

Netty 实现的 echo 服务器都需要下面这些：

- 一个服务器 handler：这个组件实现了服务器的业务逻辑，决定了连接创建后和接收到信息后该如何处理
- Bootstrapping： 这个是配置服务器的启动代码。最少需要设置服务器绑定的端口，用来监听连接请求。

通过 ChannelHandler 来实现服务器的逻辑

Echo Server 将会将接受到的数据的拷贝发送给客户端。因此，我们需要实现 ChannelInboundHandler 接口，用来定义处理入站事件的方法。由于我们的应用很简单，只需要继承 ChannelInboundHandlerAdapter 就行了。这个类 提供了默认 ChannelInboundHandler 的实现，所以只需要覆盖下面的方法：

- channelRead() - 每个信息入站都会调用
- channelReadComplete() - 通知处理器最后的 channelRead() 是当前批处理中的最后一条消息时调用
- exceptionCaught()- 读操作时捕获到异常时调用

EchoServerHandler 代码如下：

Listing 2.2 EchoServerHandler

```
@Sharable                                        //1
public class EchoServerHandler extends
        ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx,
        Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println("Server received: " + in.toString(CharsetUtil.UTF_8));
   //2
        ctx.write(in);                            //3
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        ctx.writeAndFlush(Unpooled.EMPTY_BUFFER)//4
        .addListener(ChannelFutureListener.CLOSE);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {
        cause.printStackTrace();                //5
        ctx.close();                            //6
    }
}
```

1. `@Sharable` 标识这类的实例之间可以在 channel 里面共享

2.日志消息输出到控制台

3.将所接收的消息返回给发送者。注意，这还没有冲刷数据

4.冲刷所有待审消息到远程节点。关闭通道后，操作完成

5.打印异常堆栈跟踪

6.关闭通道

这种使用 ChannelHandler 的方式体现了关注点分离的设计原则，并简化业务逻辑的迭代开发的要求。处理程序很简单;它的每一个方法可以覆盖到“hook（钩子）”在活动周期适当的点。很显然，我们覆盖 channelRead因为我们需要处理所有接收到的数据。

覆盖 exceptionCaught 使我们能够应对任何 Throwable 的子类型。在这种情况下我们记录，并关闭所有可能处于未知状态的连接。它通常是难以 从连接错误中恢复，所以干脆关闭远程连接。当然，也有可能的情况是可以从错误中恢复的，所以可以用一个更复杂的措施来尝试识别和处理 这样的情况。

如果异常没有被捕获，会发生什么？

每个 *Channel* 都有一个关联的 *ChannelPipeline*，它代表了 *ChannelHandler* 实例的链。适配器处理的实现只是将一个处理方法调用转发到链中的下一个处理器。因此，如果一个 *Netty* 应用程序不覆盖*exceptionCaught* ，那么这些错误将最终到达 *ChannelPipeline*，并且结束警告将被记录。出于这个原因，你应该提供至少一个 实现 *exceptionCaught* 的 *ChannelHandler*。

关键点要牢记：

- ChannelHandler 是给不同类型的事件调用
- 应用程序实现或扩展 ChannelHandler 挂接到事件生命周期和提供自定义应用逻辑。

引导服务器

了解到业务核心处理逻辑 EchoServerHandler 后，下面要引导服务器自身了。

- 监听和接收进来的连接请求
- 配置 Channel 来通知一个关于入站消息的 EchoServerHandler 实例

*Transport(*传输）

在本节中，你会遇到*"transport(*传输）*"*一词。在网络的多层视图协议里面，传输层提供了用于端至端或主机到主机的通信服务。互联网通信的基础是 *TCP* 传输。当我们使用术语*"NIO transport"*我们指的是一个传输的实现，它是大多等同于 *TCP* ，除了一些由 *Java NIO* 的实现提供了服务器端的性能增强。*Transport* 详细在第*4*章中讨论。

Listing 2.3 EchoServer

```
public class EchoServer {

    private final int port;

    public EchoServer(int port) {
        this.port = port;
    }
        public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println(
                    "Usage: " + EchoServer.class.getSimpleName() +
                    " <port>");
            return;
        }
        int port = Integer.parseInt(args[0]);        //1
        new EchoServer(port).start();                //2
    }

    public void start() throws Exception {
        NioEventLoopGroup group = new NioEventLoopGroup(); //3
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(group)                                //4
             .channel(NioServerSocketChannel.class)        //5
             .localAddress(new InetSocketAddress(port))    //6
             .childHandler(new ChannelInitializer<SocketChannel>() { //7
                 @Override
                 public void initChannel(SocketChannel ch)
                     throws Exception {
                     ch.pipeline().addLast(
                             new EchoServerHandler());
                 }
             });

            ChannelFuture f = b.bind().sync();            //8
            System.out.println(EchoServer.class.getName() + " started and listen on "
+ f.channel().localAddress());
            f.channel().closeFuture().sync();            //9
        } finally {
            group.shutdownGracefully().sync();            //10
        }
    }

}
```

1.设置端口值（抛出一个 NumberFormatException 如果该端口参数的格式不正确）

2.呼叫服务器的 start() 方法

3.创建 EventLoopGroup

4.创建 ServerBootstrap

5.指定使用 NIO 的传输 Channel

6.设置 socket 地址使用所选的端口

7.添加 EchoServerHandler 到 Channel 的 ChannelPipeline

8.绑定的服务器;sync 等待服务器关闭

9.关闭 channel 和 块，直到它被关闭

10.关闭 EventLoopGroup，释放所有资源。

在这个例子中，代码创建 ServerBootstrap 实例（步骤4）。由于我们使用在 NIO 传输，我们已指定 NioEventLoopGroup（3）接受和处理新连接，指定 NioServerSocketChannel（5）为信道类型。在此之后，我们设置本地地址是 InetSocketAddress 与所选择的端口（6）如。服务器将绑定到此地址来监听新的连接请求。

第7步是关键：在这里我们使用一个特殊的类，ChannelInitializer 。当一个新的连接被接受，一个新的子 Channel 将被创建， ChannelInitializer 会添加我们EchoServerHandler 的实例到 Channel 的 ChannelPipeline。正如我们如前所述，如果有入站信息，这个处理器将被通知。

虽然 NIO 是可扩展性，但它的正确配置是不简单的。特别是多线程，要正确处理也非易事。幸运的是，Netty 的设计封装了大部分复杂性，尤其是通过抽象，例如 EventLoopGroup，SocketChannel 和 ChannelInitializer，其中每一个将在更详细地在第3章中讨论。

在步骤8，我们绑定的服务器，等待绑定完成。（调用 sync() 的原因是当前线程阻塞）在第9步的应用程序将等待服务器 Channel 关闭（因为我们 在 Channel 的 CloseFuture 上调用 sync()）。现在，我们可以关闭下 EventLoopGroup 并释放所有资源，包括所有创建的线程（10）。

NIO 用于在本实施例，因为它是目前最广泛使用的传输，归功于它的可扩展性和彻底的不同步。但不同的传输的实现是也是可能的。例如，如果本实施例中使用的 OIO 传输，我们将指定 OioServerSocketChannel 和 OioEventLoopGroup。 Netty 的架构，包括更关于传输信息，将包含在第4章。在此期间，让我们回顾下在服务器上执行，我们只研究重要步骤。

服务器的主代码组件是

- EchoServerHandler 实现了的业务逻辑
- 在 main() 方法，引导了服务器

执行后者所需的步骤是：

- 创建 ServerBootstrap 实例来引导服务器并随后绑定
- 创建并分配一个 NioEventLoopGroup 实例来处理事件的处理，如接受新的连接和读/写数据。

- 指定本地 InetSocketAddress 给服务器绑定
- 通过 EchoServerHandler 实例给每一个新的 Channel 初始化
- 最后调用 ServerBootstrap.bind() 绑定服务器

这样服务器初始化完成，可以被使用了。

写一个 echo 客户端

客户端要做的是：

- 连接服务器
- 发送信息
- 发送的每个信息，等待和接收从服务器返回的同样的信息
- 关闭连接

用 ChannelHandler 实现客户端逻辑

跟写服务器一样，我们提供 ChannelInboundHandler 来处理数据。下面例子，我们用 SimpleChannelInboundHandler 来处理所有的任务，需要覆盖三个方法：

- channelActive() - 服务器的连接被建立后调用
- channelRead0() - 数据后从服务器接收到调用
- exceptionCaught() - 捕获一个异常时调用

Listing 2.4 ChannelHandler for the client

```
@Sharable                                //1
public class EchoClientHandler extends
        SimpleChannelInboundHandler<ByteBuf> {

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.copiedBuffer("Netty rocks!", //2
        CharsetUtil.UTF_8));
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
        ByteBuf in) {
        System.out.println("Client received: " + in.toString(CharsetUtil.UTF_8));    /
/3
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
        Throwable cause) {                    //4
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. `@Sharable` 标记这个类的实例可以在 channel 里共享

2.当被通知该 channel 是活动的时候就发送信息

3.记录接收到的消息

4.记录日志错误并关闭 channel

建立连接后该 channelActive() 方法被调用一次。逻辑很简单：一旦建立了连接，字节序列被发送到服务器。该消息的内容并不重要;在这里，我们使用了 Netty 编码字符串 “Netty rocks!” 通过覆盖这种方法，我们确保东西被尽快写入到服务器。

接下来，我们覆盖方法 channelRead0()。这种方法会在接收到数据时被调用。注意，由服务器所发送的消息可以以块的形式被接收。即，当服务器发送 5 个字节是不是保证所有的 5 个字节会立刻收到 - 即使是只有 5 个字节，channelRead0() 方法可被调用两次，第一次用一个 ByteBuf（Netty的字节容器）装载3个字节和第二次一个 ByteBuf 装载 2 个字节。唯一要保证的是，该字节将按照它们发送的顺序分别被接收。（注意，这是真实的，只有面向流的协议如TCP）。

第三个方法重写是 exceptionCaught()。正如在 EchoServerHandler （清单2.2），所述的记录 Throwable 并且关闭通道，在这种情况下终止 连接到服务器。

SimpleChannelInboundHandler vs. ChannelInboundHandler

何时用这两个要看具体业务的需要。在客户端，当 *channelRead0()* 完成，我们已经拿到的入站的信息。当方法返回时，*SimpleChannelInboundHandler* 会小心的释放对 *ByteBuf*（保存信息） 的引用。而在 *EchoServerHandler,*我们需要将入站的信息返回给发送者，由于 *write()* 是异步的，在 *channelRead()* 返回时，可能还没有完成。所以，我们使用 *ChannelInboundHandlerAdapter,*无需释放信息。最后在 *channelReadComplete()* 我们调用 *ctxWriteAndFlush()* 来释放信息。详见第*5*、*6*章

引导客户端

客户端引导需要 host 、port 两个参数连接服务器。

Listing 2.5 Main class for the client

```
public class EchoClient {

    private final String host;
    private final int port;

    public EchoClient(String host, int port) {
        this.host = host;
        this.port = port;
    }

    public void start() throws Exception {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap b = new Bootstrap();                //1
            b.group(group)                                //2
             .channel(NioSocketChannel.class)            //3
             .remoteAddress(new InetSocketAddress(host, port))    //4
             .handler(new ChannelInitializer<SocketChannel>() {    //5
                @Override
                public void initChannel(SocketChannel ch)
                    throws Exception {
                    ch.pipeline().addLast(
                            new EchoClientHandler());
                }
             });

            ChannelFuture f = b.connect().sync();        //6

            f.channel().closeFuture().sync();            //7
        } finally {
            group.shutdownGracefully().sync();            //8
        }
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println(
                    "Usage: " + EchoClient.class.getSimpleName() +
                    " <host> <port>");
            return;
        }

        final String host = args[0];
        final int port = Integer.parseInt(args[1]);

        new EchoClient(host, port).start();
    }
}
```

1.创建 Bootstrap

2.指定 EventLoopGroup 来处理客户端事件。由于我们使用 NIO 传输，所以用到了 NioEventLoopGroup 的实现

3.使用的 channel 类型是一个用于 NIO 传输

4.设置服务器的 InetSocketAddress

5.当建立一个连接和一个新的通道时，创建添加到 EchoClientHandler 实例 到 channel pipeline

6.连接到远程;等待连接完成

7.阻塞直到 Channel 关闭

8.调用 shutdownGracefully() 来关闭线程池和释放所有资源

与以前一样，在这里使用了 NIO 传输。请注意，您可以在 客户端和服务器 使用不同的传输，例如 NIO 在服务器端和 OIO 客户端。在第四章中，我们将研究一些具体的因素和情况，这将导致 您可以选择一种传输，而不是另一种。

让我们回顾一下我们在本节所介绍的要点

- 一个 Bootstrap 被创建来初始化客户端
- 一个 NioEventLoopGroup 实例被分配给处理该事件的处理，这包括创建新的连接和处理入站和出站数据
- 一个 InetSocketAddress 为连接到服务器而创建
- 一个 EchoClientHandler 将被安装在 pipeline 当连接完成时
- 之后 Bootstrap.connect（）被调用连接到远程的 - 本例就是 echo(回声)服务器。

编译和运行 Echo 服务器和客户端

编译

本例涉及到多模块 *Maven* 项目的组织

在例子 chapter2 目录下，执行

```
mvn clean package
```

输出如下

Listing 2.6 Build Output

```
chapter2>mvn clean package
[INFO] Scanning for projects...
[INFO] --------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Echo Client and Server
[INFO] Echo Client
[INFO] Echo Server
[INFO]
[INFO] --------------------------------------------------------------------
[INFO] Building Echo Client and Server 1.0-SNAPSHOT
[INFO] --------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ echo-parent ---
[INFO]
[INFO] --------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] --------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ echo-client ---
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile)
@ echo-client ---
[INFO] Changes detected - recompiling the module!
[INFO] --------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Echo Client and Server ......................... SUCCESS [ 0.118 s]
[INFO] Echo Client .................................... SUCCESS [ 1.219 s]
[INFO] Echo Server .................................... SUCCESS [ 0.110 s]
[INFO] --------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] --------------------------------------------------------------------
[INFO] Total time: 1.561 s
[INFO] Finished at: 2014-06-08T17:39:15-05:00
[INFO] Final Memory: 14M/245M
[INFO] --------------------------------------------------------------------
```

注意事项：

- Maven Reactor 构建顺序：先是 父 POM，然后是子项目
- Netty artifact 没在用户的本地存储库中找到，所以 Maven 就会从互联网上下载
- clean 和 compile 在构建生命周期的运行。事后 mavensurefire-plugin 插件运行，但不会有测试类存在。最后 mavenjar-plugin 执行

这段说明了项目已经成功编译。

运行 Echo 服务器 和 客户端

我们使用 exec-maven-plugin 来运行项目。

在 chapter2/Server 目录，执行

```
mvn exec:java
```

输出如下：

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------------------------------------------------
[INFO] Building Echo Server 1.0-SNAPSHOT
[INFO] -------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-server >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-server <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-server ---
nettyinaction.echo.EchoServer started and listening for connections on
/0:0:0:0:0:0:0:0:9999
```

在 chapter2/Client 目录，执行

```
mvn exec:java
```

输出如下：

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] -------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-client >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-client <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---
Client received: Netty rocks!
[INFO] -------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------------------
[INFO] Total time: 3.907 s
[INFO] Finished at: 2014-06-08T18:26:14-05:00
[INFO] Final Memory: 8M/245M
[INFO] -------------------------------------------------------------------
```

在服务器的控制台输出：

```
Server received: Netty rocks!
```

发生了什么事：

- 客户端连接后，它发送消息：“Netty rocks！”
- 服务器输出接收到消息并将其返回给客户端
- 客户输出接收到的消息并退出。

每次运行客户端，你会看到在服务器的控制台输出：

```
Server received: Netty rocks!
```

现在，我们看下错误的情况。在控制台 输入 Ctrl-C 来关闭服务器。而后运行客户端，此时输出如下：

```
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------------------------------------------------------
[INFO] Building Echo Client 1.0-SNAPSHOT
[INFO] --------------------------------------------------------------------
[INFO]
[INFO] >>> exec-maven-plugin:1.2.1:java (default-cli) @ echo-client >>>
[INFO]
[INFO] <<< exec-maven-plugin:1.2.1:java (default-cli) @ echo-client <<<
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ echo-client ---
[WARNING]
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke
    (NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke
    (DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:606)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:297)
    at java.lang.Thread.run(Thread.java:744)
    Caused by: java.net.ConnectException: Connection refused:
    no further information: localhost/127.0.0.1:9999
    at sun.nio.ch.SocketChannelImpl.checkConnect(Native Method)
    at sun.nio.ch.SocketChannelImpl.finishConnect
    (SocketChannelImpl.java:739)
    at io.netty.channel.socket.nio.NioSocketChannel
    .doFinishConnect(NioSocketChannel.java:191)
    at io.netty.channel.nio.
    AbstractNioChannel$AbstractNioUnsafe.finishConnect(
    AbstractNioChannel.java:279)
```

```
    at io.netty.channel.nio.NioEventLoop
    .processSelectedKey(NioEventLoop.java:511)
    at io.netty.channel.nio.NioEventLoop
    .processSelectedKeysOptimized(NioEventLoop.java:461)
    at io.netty.channel.nio.NioEventLoop
    .processSelectedKeys(NioEventLoop.java:378)
    at io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:350)
    at io.netty.util.concurrent
    .SingleThreadEventExecutor$2.run
    (SingleThreadEventExecutor.java:101)
    ... 1 more
[INFO] --------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] --------------------------------------------------------------------
[INFO] Total time: 3.728 s
[INFO] Finished at: 2014-06-08T18:49:13-05:00
[INFO] Final Memory: 8M/245M
[INFO] --------------------------------------------------------------------
[ERROR] Failed to execute goal org.codehaus.mojo:exec-maven-plugin:1.2.1:java
    (default-cli) on project echo-client: An exception occured while executing the
    Java class. null: InvocationTargetException: Connection refused: no further
    information:
    localhost/127.0.0.1:9999 -> [Help 1]
```

发生了啥？客户端尝试连接服务器，但服务器是关闭的，所以引发了一个 java.net.ConnectException ，这个异常被 EchoClientHandler 的 exceptionCaught() 触发，打印出异常信息，并关闭 channel

总结

在本章中，您构建并运行你的第一 个Netty 的客户端和服务器。虽然这是一个简单的应用程序，它可以扩展到几千个并发连接。

在下面的章节中，我们会看到的更多 Netty 如何简化可扩展和多线程的例子。我们还将更深入的了解 Netty 支持的关注点分离的构建理念;通过提供正确的抽象将业务逻辑从网络逻辑中解耦，Netty 可以很容易地跟上迅速发展的要求，而不损害系统的稳定性。

在下一章中，我们将提供的 Netty 的架构的概述。

Netty 总览

本章主要了解 Netty 的架构模型，核心组件包括：

- Bootstrap 和 ServerBootstrap
- Channel
- ChannelHandler
- ChannelPipeline
- EventLoop
- ChannelFuture

这个目标是提供一个深入研究的上下文，如果你有一个很好的把握它 组织原则，可以避免迷失。

Netty 快速入门

下面枚举所有的 Netty 应用程序的基本构建模块，包括客户端和服务器。

BOOTSTRAP

Netty 应用程序通过设置 bootstrap（引导）类的开始，该类提供了一个 用于应用程序网络层配置的容器。

CHANNEL

底层网络传输 API 必须提供给应用 I/O操作的接口，如读，写，连接，绑定等等。对于我们来说，这是结构几乎总是会成为一个“socket”。 Netty 中的接口 Channel 定义了与 socket 丰富交互的操作集：bind, close, config, connect, isActive, isOpen, isWritable, read, write 等等。 Netty 提供大量的 Channel 实现来专门使用。这些包括 AbstractChannel，AbstractNioByteChannel，AbstractNioChannel，EmbeddedChannel，LocalServerChannel，NioSocketChannel 等等。

CHANNELHANDLER

ChannelHandler 支持很多协议，并且提供用于数据处理的容器。我们已经知道 ChannelHandler 由特定事件触发。 ChannelHandler 可专用于几乎所有的动作，包括将一个对象转为字节（或相反），执行过程中抛出的异常处理。

常用的一个接口是 ChannelInboundHandler，这个类型接收到入站事件（包括接收到的数据）可以处理应用程序逻辑。当你需要提供响应时，你也可以从 ChannelInboundHandler 冲刷数据。一句话，业务逻辑经常存活于一个或者多个 ChannelInboundHandler。

CHANNELPIPELINE

ChannelPipeline 提供了一个容器给 ChannelHandler 链并提供了一个API 用于管理沿着链入站和出站事件的流动。每个 Channel 都有自己的ChannelPipeline，当 Channel 创建时自动创建的。 ChannelHandler 是如何安装在 ChannelPipeline？ 主要是实现了ChannelHandler 的抽象 ChannelInitializer。ChannelInitializer子类 通过 ServerBootstrap 进行注册。当它的方法 initChannel() 被调用时，这个对象将安装自定义的 ChannelHandler 集到 pipeline。当这个操作完成时，ChannelInitializer 子类则 从 ChannelPipeline 自动删除自身。

EVENTLOOP

EventLoop 用于处理 Channel 的 I/O 操作。一个单一的 EventLoop通常会处理多个 Channel 事件。一个 EventLoopGroup 可以含有多于一个的 EventLoop 和 提供了一种迭代用于检索清单中的下一个。

CHANNELFUTURE

Netty 所有的 I/O 操作都是异步。因为一个操作可能无法立即返回，我们需要有一种方法在以后确定它的结果。出于这个目的，Netty 提供了接口 ChannelFuture,它的 addListener 方法注册了一个 ChannelFutureListener ，当操作完成时，可以被通知（不管成功与否）。

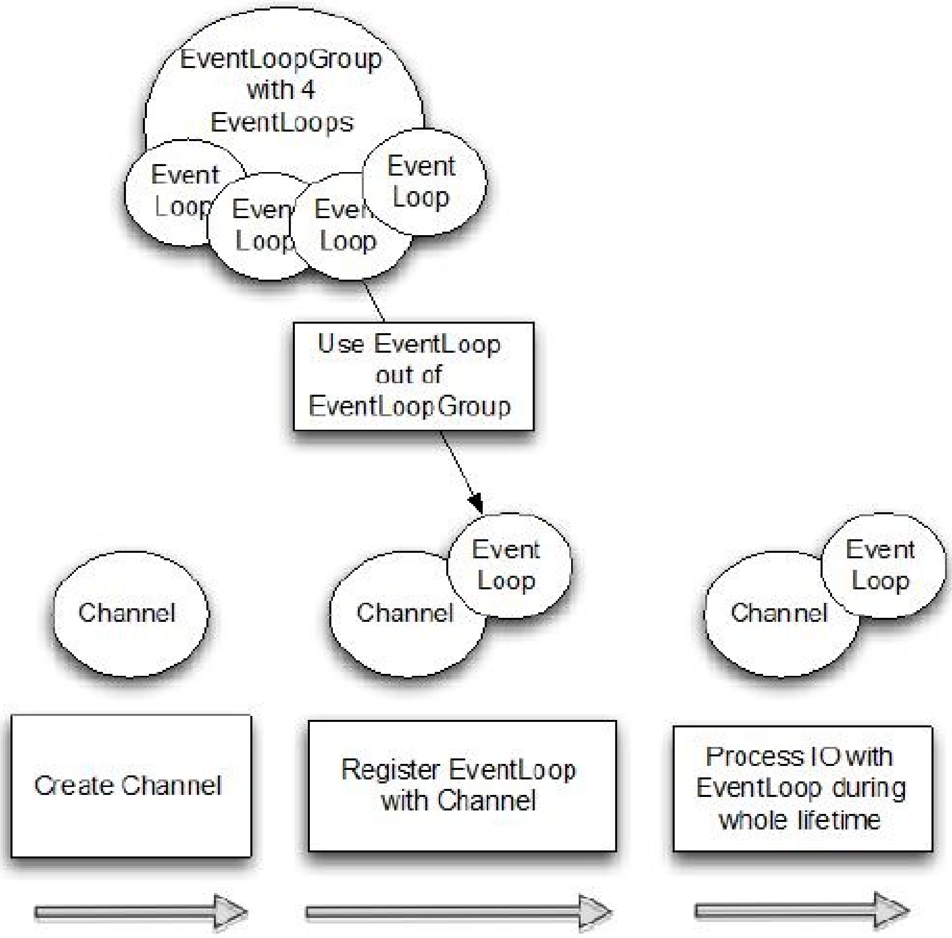
更多关于 *ChannelFuture*

想想一个 *ChannelFuture* 对象作为一个未来执行操作结果的占位符。何时执行取决于几个因素，因此不可能预测与精确。但我们可以肯定的是，它会被执行。此外，所有的操作返回 *ChannelFuture* 对象和属于同一个 *Channel* 将在以正确的顺序被执行，在他们被调用后。

Channel, Event 和 I/O

Netty 是一个非阻塞、事件驱动的网络框架。Netty 实际上是使用 Threads（多线程）处理 I/O 事件，对于熟悉多线程编程的读者可能会需要关注同步代码。这样的方式不好，因为同步会影响程序的性能，Netty 的设计保证程序处理事件不会有同步。图 Figure 3.1 展示了，你不需要在 Channel 之间共享 ChannelHandler 实例的原因：

Figure 3.1



该图显示，一个 EventLoopGroup 具有一个或多个 EventLoop。想象 EventLoop 作为一个 Thread 给 Channel 执行工作。（事实上，一个 EventLoop 是势必为它的生命周期一个线程。）

当创建一个 Channel，Netty 通过 一个单独的 EventLoop 实例来注册该 Channel（并同样是一个单独的 Thread）的通道的使用寿命。这就是为什么你的应用程序不需要同步 Netty 的 I/O操作;所有 Channel 的 I/O 始终用相同的线程来执行。

我们将在第15章进一步讨论 EventLoop 和 EventLoopGroup。

什么是 Bootstrapping 为什么要用

Bootstrapping（引导） 是 Netty 中配置程序的过程，当你需要连接客户端或服务器绑定指定端口时需要使用 Bootstrapping。

如前面所述，Bootstrapping 有两种类型，一种是用于客户端的Bootstrap，一种是用于服务端的ServerBootstrap。不管程序使用哪种协议，无论是创建一个客户端还是服务器都需要使用“引导”。

面向连接 *vs.* 无连接

请记住，这个讨论适用于 *TCP* 协议，它是“面向连接”的。这样协议保证该连接的端点之间的消息的有序输送。无连接协议发送的消息，无法保证顺序和成功性

两种 Bootstrapping 之间有一些相似之处，也有一些不同。Bootstrap 和 ServerBootstrap 之间的差异如下：

Table 3.1 Comparison of Bootstrap classes

分类	Bootstrap	ServerBootstrap
网络功能	连接到远程主机和端口	绑定本地端口
EventLoopGroup 数量	1	2

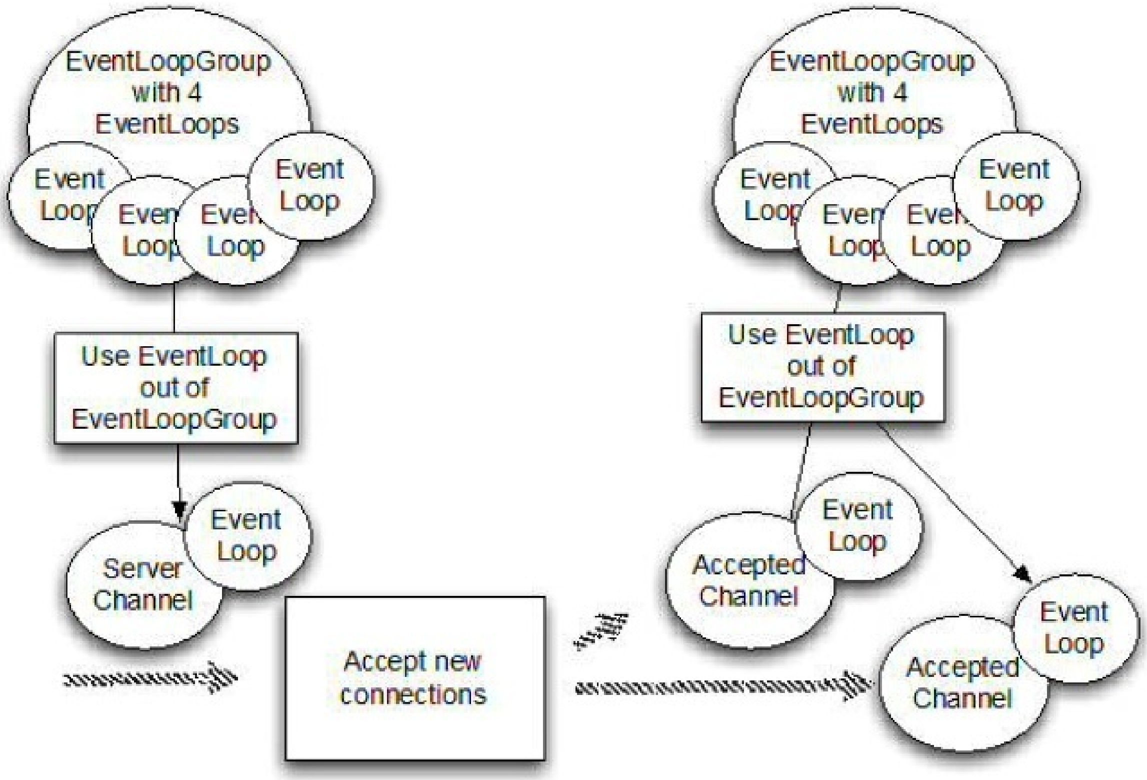
Bootstrap用来连接远程主机，有1个EventLoopGroup

ServerBootstrap用来绑定本地端口，有2个EventLoopGroup

事件组(Groups)，传输(transports)和处理程序(handlers)分别在本章后面讲述，我们在这里只讨论两种"引导"的差异(Bootstrap和ServerBootstrap)。第一个差异很明显，“ServerBootstrap”监听在服务器监听一个端口轮询客户端的“Bootstrap”或DatagramChannel是否连接服务器。通常需要调用“Bootstrap”类的connect()方法，但是也可以先调用bind()再调用connect()进行连接，之后使用的Channel包含在bind()返回的ChannelFuture中。

一个 ServerBootstrap 可以认为有2个 Channel 集合，第一个集合包含一个单例 ServerChannel，代表持有一个绑定了本地端口的 socket；第二集合包含所有创建的 Channel，处理服务器所接收到的客户端进来的连接。下图形象的描述了这种情况：

Figure 3.2 Server with two EventLoopGroups



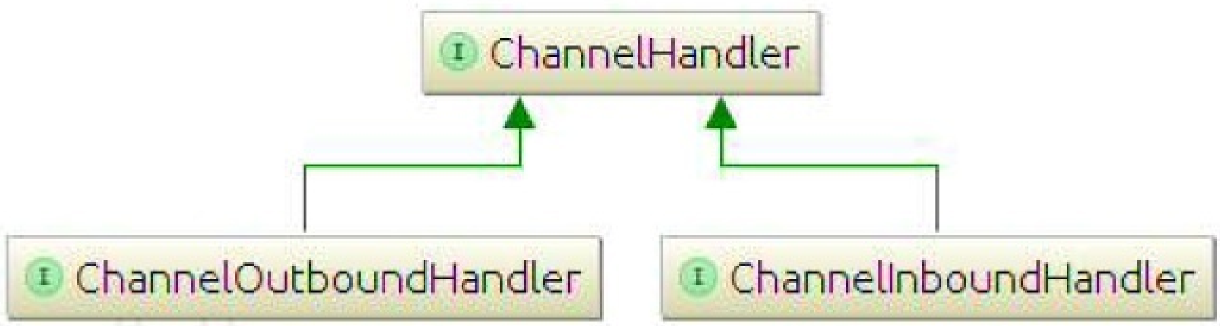
与 ServerChannel 相关 EventLoopGroup 分配一个 EventLoop 是 负责创建 Channels 用于传入的连接请求。一旦连接接受，第二个EventLoopGroup 分配一个 EventLoop 给它的 Channel。

ChannelHandler 和 ChannelPipeline

ChannelPipeline 是 ChannelHandler 链的容器。

在许多方面的 ChannelHandler 是在您的应用程序的核心，尽管有时它 可能并不明显。ChannelHandler 支持广泛的用途，使它难以界定。因此，最好是把它当作一个通用的容器，处理进来的事件（包括数据）并且通过ChannelPipeline。下图展示了 ChannelInboundHandler 和 ChannelOutboundHandler 继承自父接口 ChannelHandler。

Figure 3.3 ChannelHandler class hierarchy



Netty 中有两个方向的数据流，图3.4 显示的入站(ChannelInboundHandler)和出站(ChannelOutboundHandler)之间有一个明显的区别：若数据是从用户应用程序到远程主机则是“出站(outbound)”，相反若数据时从远程主机到用户应用程序则是“入站(inbound)”。

为了使数据从一端到达另一端，一个或多个 ChannelHandler 将以某种方式操作数据。这些 ChannelHandler 会在程序的“引导”阶段被添加ChannelPipeline中，并且被添加的顺序将决定处理数据的顺序。

Figure 3.4 ChannelPipeline with inbound and outbound ChannelHandlers

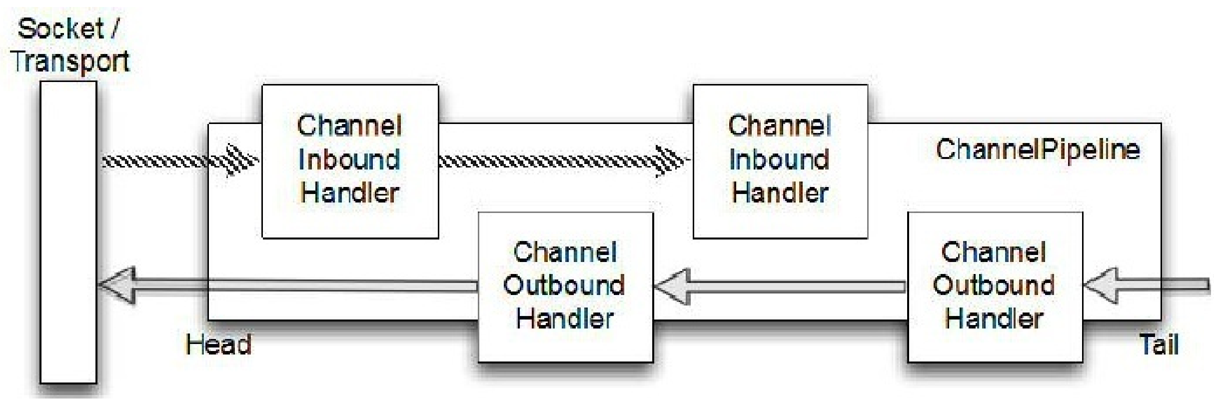


图 3.4 同样展示了进站和出站的处理器都可以被安装在相同的 pipeline 。本例子中，如果消息或任何其他入站事件被读到，将从 pipeline 头部开始，传递到第一个 ChannelInboundHandler。该处理器可能会或可能不会实际修改数据，取决于其特定的功能，在这之后 该数据将被传递到链中的下一个 ChannelInboundHandler。最后，将数据 到达 pipeline 的尾部，此时所有处理结束。

数据的出站运动（即，数据被“写入”）在概念上是相同的。在这种情况下的数据从尾部流过 ChannelOutboundHandlers 的链，直到它到达头部。超过这点，出站数据将到达的网络传输，在这里显示为一个 socket。通常，这将触发一个写入操作。

更多 *Inbound* 、 *Outbound Handler*

在当前的链（*chain*）中，事件可以通过 *ChanneHandlerContext* 传递给下一个 *handler*。*Netty* 为此提供了抽象基类*ChannelInboundHandlerAdapter* 和 *hannelOutboundHandlerAdapter*，用来处理你想要的事件。 这些类提供的方法的实现，可以简单地通过调用 *ChannelHandlerContext* 上的相应方法将事件传递给下一个 *handler*。在实际应用中，您可以按需覆盖相应的方法即可。

所以，如果出站和入站操作是不同的，当 ChannelPipeline 中有混合处理器时将发生什么？虽然入站和出站处理器都扩展了 ChannelHandler，Netty 的 ChannelInboundHandler 的实现 和 ChannelOutboundHandler 之间的是有区别的，从而保证数据传递只从一个处理器到下一个处理器保证正确的类型。

当 ChannelHandler 被添加到的 ChannelPipeline 它得到一个 ChannelHandlerContext，它代表一个 ChannelHandler 和 ChannelPipeline 之间的“绑定”。它通常是安全保存对此对象的引用，除了当协议中的使用的是不面向连接（例如，UDP）。而该对象可以被用来获得 底层 Channel,它主要是用来写出站数据。

还有，实际上，在 Netty 发送消息有两种方式。您可以直接写消息给 Channel 或写入 ChannelHandlerContext 对象。主要的区别是， 前一种方法会导致消息从 ChannelPipeline的尾部开始，而 后者导致消息从 ChannelPipeline 下一个处理器开始。

近距离观察 ChannelHandler

正如我们之前所说，有很多不同类型的 ChannelHandler 。每个 ChannelHandler 做什么取决于其超类。 Netty 提供了一些默认的处理程序实现形式的“adapter（适配器）”类。这些旨在简化开发处理逻辑。我们已经看到，在 pipeline 中每个的 ChannelHandler 负责转发事件到链中的下一个处理器。这些适配器类（及其子类）会自动帮你实现，所以你只需要实现该特定的方法和事件。

为什么用适配器？

有几个适配器类，可以减少编写自定义 *ChannelHandlers* ，因为他们提供对应接口的所有方法的默认实现。（也有类似的适配器，用于创建编码器和解码器，这我们将在稍后讨论。）这些都是创建自定义处理器时，会经常调用的适配器：*ChannelHandlerAdapter*、*ChannelInboundHandlerAdapter*、*ChannelOutboundHandlerAdapter*、*ChannelDuplexHandlerAdapter*

下面解释下三个 ChannelHandler 的子类型：编码器、解码器以及 ChannelInboundHandlerAdapter 的子类SimpleChannelInboundHandler

编码器、解码器

当您发送或接收消息时，Netty 数据转换就发生了。入站消息将从字节转为一个Java对象;也就是说，“解码”。如果该消息是出站相反会发生：“编码”，从一个Java对象转为字节。其原因是简单的：网络数据是一系列字节，因此需要从那类型进行转换。

不同类型的抽象类用于提供编码器和解码器的，这取决于手头的任务。例如，应用程序可能并不需要马上将消息转为字节。相反，该消息将被转换 一些其他格式。一个编码器将仍然可以使用，但它也将衍生自不同的超类，

在一般情况下，基类将有一个名字类似 ByteToMessageDecoder 或 MessageToByteEncoder。在一种特殊类型的情况下，你可能会发现类似 ProtobufEncoder 和 ProtobufDecoder，用于支持谷歌的 protocol buffer。

严格地说，其他处理器可以做编码器和解码器能做的事。但正如适配器类简化创建通道处理器，所有的编码器/解码器适配器类 都实现自 ChannelInboundHandler 或 ChannelOutboundHandler。

对于入站数据，channelRead 方法/事件被覆盖。这种方法在每个消息从入站 Channel 读入时调用。该方法将调用特定解码器的“解码”方法，并将解码后的消息转发到管道中下个的 ChannelInboundHandler。

出站消息是类似的。编码器将消息转为字节，转发到下个的 ChannelOutboundHandler。

SimpleChannelHandler

也许最常见的处理器是接收到解码后的消息并应用一些业务逻辑到这些数据。要创建这样一个 ChannelHandler，你只需要扩展基类SimpleChannelInboundHandler 其中 T 是想要进行处理的类型。这样的处理器，你将覆盖基类的一个或多个方法，将获得被作为输入参数传递所有方法的 ChannelHandlerContext 的引用。

在这种类型的处理器方法中的最重要是 channelRead0(ChannelHandlerContext，T)。在这个调用中，T 是将要处理的消息。 你怎么做，完全取决于你，但无论如何你不能阻塞 I/O线程，因为这可能是不利于高性能。

阻塞操作

I/O 线程一定不能完全阻塞，因此禁止任何直接阻塞操作在你的 *ChannelHandler*， 有一种方法来实现这一要求。你可以指定一个 *EventExecutorGroup* 当添加 *ChannelHandler* 到 *ChannelPipeline*。此 *EventExecutorGroup* 将用于获得*EventExecutor*，将执行所有的 *ChannelHandler* 的方法。这*EventExecutor* 将从 *I/O* 线程使用不同的线程，从而释放 *EventLoop*。

总结

在本章中，我们提出了 Netty 的关键部件和概念的概述，以及他们是如何结合在一起的。许多下面的章节都致力于深入研究各个组件和概念，应该可以帮助你了解全貌。

下一章将探讨 Netty 并提供不同的传输，以及如何选择最适合您应用程序的传输。

Transport

本章将涵盖很多 transport(传输)，他们的用例以及 API:

- NIO
- OIO
- Local(本地)
- Embedded(内嵌)

网络应用程序提供了人与系统通信的信道，但是，当然 他们也将大量的数据从一个地方移到到另一个地方。如何做到这一点取决于具体的网络传输，但转移始终是相同的：字节通过线路。传输的概念帮助我们抽象掉的底层数据转移的机制。所有人都需要知道的是，字节在被发送和接收。

如果你已经做过 Java中 的网络编程，你可能会发现在某些时候你必须支持比预期更多的并发连接。如果你再尝试从阻塞切换到非阻塞传输，则可能遇会到的问题，因为 Java 的公开的网络 API 来处理这两种情况有很大的不同。

Netty 在传输层是统一的API，这使得比你用 JDK 实现更简单。你无需重构整个代码库。总之，你可以省下时间去做其他更富有成效的事。

在本章中，我们将研究这个统一的 API，与 JDK 进行演示对比，可以见它具有更大的易用性。我们将介绍不同的 捆绑在 Netty 的传输实现和适当的的用例。吸收这些信息后，你就知道如何选择适合您的应用的最佳选择。

本章的唯一前提是 Java 编程语言的知识。 最好是有网络框架或网络编程的经验，但也不是必需的。

让我们看看现实世界传输是如何工作的。

案例研究:Transport 的迁移

为了让你想象 Transport 如何工作，我会从一个简单的应用程序开始，这个应用程序什么都不做，只是接受客户端连接并发送“Hi!”字符串消息到客户端，发送完了就断开连接。

没有用 Netty 实现 I/O 和 NIO

我们将不用 Netty 实现只用 JDK API 来实现 I/O 和 NIO。下面这个例子，是使用阻塞 IO 实现的例子：

Listing 4.1 Blocking networking without Netty

```
public class PlainOioServer {

    public void serve(int port) throws IOException {
        final ServerSocket socket = new ServerSocket(port);     //1
        try {
            for (;;) {
                final Socket clientSocket = socket.accept();    //2
                System.out.println("Accepted connection from " + clientSocket);

                new Thread(new Runnable() {                         //3
                    @Override
                    public void run() {
                        OutputStream out;
                        try {
                            out = clientSocket.getOutputStream();
                            out.write("Hi!\r\n".getBytes(Charset.forName("UTF-8")));
                          //4
                            out.flush();
                            clientSocket.close();                //5

                        } catch (IOException e) {
                            e.printStackTrace();
                            try {
                                clientSocket.close();
                            } catch (IOException ex) {
                                // ignore on close
                            }
                        }
                    }
                }).start();                                         //6
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

1.绑定服务器到指定的端口。

2.接受一个连接。

3.创建一个新的线程来处理连接。

4.将消息发送到连接的客户端。

5.一旦消息被写入和刷新时就 关闭连接。

6.启动线程。

上面的方式可以工作正常，但是这种阻塞模式在大连接数的情况就会有很严重的问题，如客户端连接超时，服务器响应严重延迟，性能无法扩展。为了解决这种情况，我们可以使用异步网络处理所有的并发连接，但问题在于 NIO 和 OIO 的 API 是完全不同的，所以一个用OIO开发的网络应用程序想要使用NIO重构代码几乎是重新开发。

下面代码是使用 NIO 实现的例子：

Listing 4.2 Asynchronous networking without Netty

```
public class PlainNioServer {
    public void serve(int port) throws IOException {
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        serverChannel.configureBlocking(false);
        ServerSocket ss = serverChannel.socket();
        InetSocketAddress address = new InetSocketAddress(port);
        ss.bind(address);                                            //1
        Selector selector = Selector.open();                        //2
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);    //3
        final ByteBuffer msg = ByteBuffer.wrap("Hi!\r\n".getBytes());
        for (;;) {
            try {
                selector.select();                                   //4
            } catch (IOException ex) {
                ex.printStackTrace();
                // handle exception
                break;
            }
            Set<SelectionKey> readyKeys = selector.selectedKeys();    //5
            Iterator<SelectionKey> iterator = readyKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                try {
                    if (key.isAcceptable()) {                //6
                        ServerSocketChannel server =
                                (ServerSocketChannel)key.channel();
                        SocketChannel client = server.accept();
                        client.configureBlocking(false);
                        client.register(selector, SelectionKey.OP_WRITE |
                                SelectionKey.OP_READ, msg.duplicate());    //7
                        System.out.println(
                                "Accepted connection from " + client);
                    }
                    if (key.isWritable()) {                //8
                        SocketChannel client =
                                (SocketChannel)key.channel();
                        ByteBuffer buffer =
                                (ByteBuffer)key.attachment();
                        while (buffer.hasRemaining()) {
                            if (client.write(buffer) == 0) {        //9
                                break;
```

```
                            }
                        }
                        client.close();                    //10
                    }
                } catch (IOException ex) {
                    key.cancel();
                    try {
                        key.channel().close();
                    } catch (IOException cex) {
                        // 在关闭时忽略
                    }
                }
            }
        }
    }
}
```

1.绑定服务器到制定端口

2.打开 selector 处理 channel

3.注册 ServerSocket 到 ServerSocket ，并指定这是专门意接受 连接。

4.等待新的事件来处理。这将阻塞，直到一个事件是传入。

5.从收到的所有事件中 获取 SelectionKey 实例。

6.检查该事件是一个新的连接准备好接受。

7.接受客户端，并用 selector 进行注册。

8.检查 socket 是否准备好写数据。

9.将数据写入到所连接的客户端。如果网络饱和，连接是可写的，那么这个循环将写入数据，直到该缓冲区是空的。

10.关闭连接。

如你所见，即使它们实现的功能是一样，但是代码完全不同。下面我们将用Netty 来实现相同的功能。

采用 Netty 实现 I/O 和 NIO

下面代码是使用Netty作为网络框架编写的一个阻塞 IO 例子：

Listing 4.3 Blocking networking with Netty

```
public class NettyOioServer {

    public void server(int port) throws Exception {
        final ByteBuf buf = Unpooled.unreleasableBuffer(
                Unpooled.copiedBuffer("Hi!\r\n", Charset.forName("UTF-8")));
        EventLoopGroup group = new OioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();        //1

            b.group(group)                                    //2
             .channel(OioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(port))
             .childHandler(new ChannelInitializer<SocketChannel>() {//3
                 @Override
                 public void initChannel(SocketChannel ch)
                     throws Exception {
                     ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {
     //4
                         @Override
                         public void channelActive(ChannelHandlerContext ctx) throws E
xception {
                             ctx.writeAndFlush(buf.duplicate()).addListener(ChannelFut
ureListener.CLOSE);//5
                         }
                     });
                 }
             });
            ChannelFuture f = b.bind().sync();  //6
            f.channel().closeFuture().sync();
        } finally {
            group.shutdownGracefully().sync();        //7
        }
    }
}
```

1.创建一个 ServerBootstrap

2.使用 OioEventLoopGroup 允许阻塞模式

3.指定 ChannelInitializer 将给每个接受的连接调用

4.添加的 ChannelHandler 拦截事件，并允许他们作出反应

5.写信息到客户端，并添加 ChannelFutureListener 当一旦消息写入就关闭连接

6.绑定服务器来接受连接

7.释放所有资源

下面代码是使用 Netty NIO 实现。

Netty NIO 版本

下面是 Netty NIO 的代码，只是改变了一行代码，就从 OIO 传输 切换到了 NIO。

Listing 4.4 Asynchronous networking with Netty

```
public class NettyNioServer {

    public void server(int port) throws Exception {
        final ByteBuf buf = Unpooled.unreleasableBuffer(
                Unpooled.copiedBuffer("Hi!\r\n", Charset.forName("UTF-8")));
        NioEventLoopGroup bossGroup = new NioEventLoopGroup(1);
    NioEventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();    //1
            b.group(bossGroup, workerGroup)   //2
             .channel(NioServerSocketChannel.class)
             .localAddress(new InetSocketAddress(port))
             .childHandler(new ChannelInitializer<SocketChannel>() {    //3
                 @Override
                 public void initChannel(SocketChannel ch)
                     throws Exception {
                     ch.pipeline().addLast(new ChannelInboundHandlerAdapter() {    //4
                         @Override
                         public void channelActive(ChannelHandlerContext ctx) throws E
xception {
                             ctx.writeAndFlush(buf.duplicate())                //5
                                .addListener(ChannelFutureListener.CLOSE);
                         }
                     });
                 }
             });
            ChannelFuture f = b.bind().sync();                    //6
            f.channel().closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully().sync();                    //7
        workerGroup.shutdownGracefully().sync();
        }
    }
}
```

1.创建一个 ServerBootstrap

2.使用 NioEventLoopGroup 允许非阻塞模式（NIO）

3.指定 ChannelInitializer 将给每个接受的连接调用

4.添加的 ChannelInboundHandlerAdapter() 接收事件并进行处理

5.写信息到客户端，并添加 ChannelFutureListener 当一旦消息写入就关闭连接

6.绑定服务器来接受连接

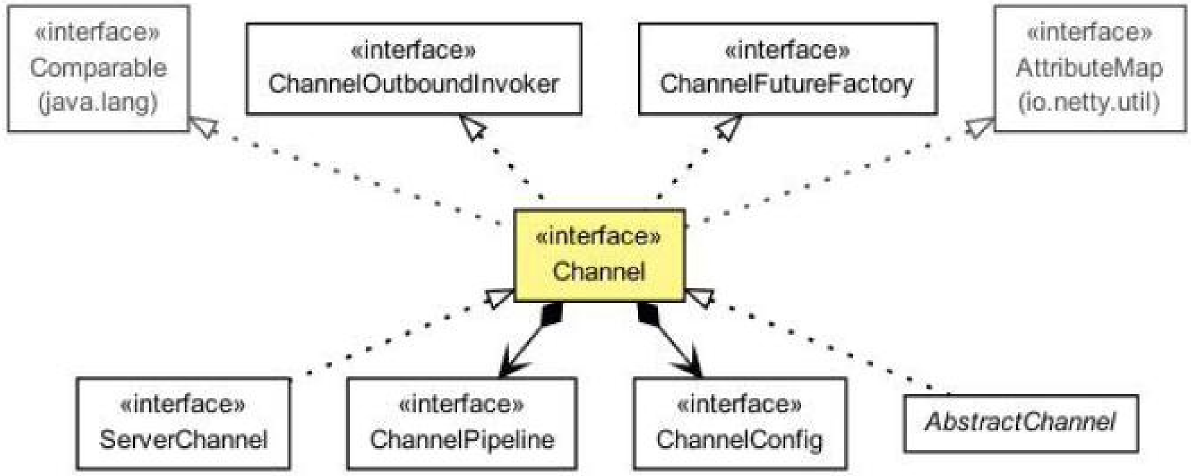
7.释放所有资源

因为 Netty 使用相同的 API 来实现每个传输，它并不关心你使用什么来实现。Netty 通过操作接口Channel 、ChannelPipeline 和 ChannelHandler来实现。

现在你了解到了用 基于 Netty 传输的好处。下面就来看下传输的 API.

Transport API

Transport API 的核心是 Channel 接口，用于所有的出站操作，见下图



如上图所示，每个 Channel 都会分配一个 ChannelPipeline 和ChannelConfig。ChannelConfig 负责设置并存储 Channel 的配置，并允许在运行期间更新它们。传输一般有特定的配置设置，可能实现了 ChannelConfig. 的子类型。

ChannelPipeline 容纳了使用的 ChannelHandler 实例，这些ChannelHandler 将处理通道传递的“入站”和“出站”数据以及事件。ChannelHandler 的实现允许你改变数据状态和传输数据。

现在我们可以使用 ChannelHandler 做下面一些事情：

- 传输数据时，将数据从一种格式转换到另一种格式
- 异常通知
- Channel 变为 active（活动） 或 inactive（非活动） 时获得通知* Channel 被注册或注销时从 EventLoop 中获得通知
- 通知用户特定事件

Intercepting Filter（拦截过滤器）

ChannelPipeline 实现了常用的 *Intercepting Filter*（拦截过滤器）设计模式。*UNIX*管道是另一例子：命令链接在一起，一个命令的输出连接到 的下一行中的输入。

你还可以在运行时根据需要添加 ChannelHandler 实例到ChannelPipeline 或从 ChannelPipeline 中删除，这能帮助我们构建高度灵活的 Netty 程序。例如，你可以支持 STARTTLS 协议，只需通过加入适当的 ChannelHandler（这里是 SslHandler）到的 ChannelPipeline 中，当被请求这个协议时。

此外，访问指定的 ChannelPipeline 和 ChannelConfig，你能在Channel 自身上进行操作。Channel 提供了很多方法，如下列表：

Table 4.1 Channel main methods

方法名称	描述
eventLoop()	返回分配给Channel的EventLoop
pipeline()	返回分配给Channel的ChannelPipeline
isActive()	返回Channel是否激活，已激活说明与远程连接对等
localAddress()	返回已绑定的本地SocketAddress
remoteAddress()	返回已绑定的远程SocketAddress
write()	写数据到远程客户端，数据通过ChannelPipeline传输过去
flush()	刷新先前的数据
writeAndFlush(...)	一个方便的方法用户调用write(...)而后调用 flush()

后面会越来越熟悉这些方法，现在只需要记住我们的操作都是在相同的接口上运行，Netty 的高灵活性让你可以以不同的传输实现进行重构。

写数据到远程已连接客户端可以调用Channel.write()方法，如下代码：

Listing 4.5 Writing to a channel

```
Channel channel = ...; // 获取channel的引用
ByteBuf buf = Unpooled.copiedBuffer("your data", CharsetUtil.UTF_8);            //1
ChannelFuture cf = channel.writeAndFlush(buf); //2

cf.addListener(new ChannelFutureListener() {    //3
    @Override
    public void operationComplete(ChannelFuture future) {
        if (future.isSuccess()) {                //4
            System.out.println("Write successful");
        } else {
            System.err.println("Write error");    //5
            future.cause().printStackTrace();
        }
    }
});
```

1.创建 ByteBuf 保存写的数据

2.写数据，并刷新

3.添加 ChannelFutureListener 即可写操作完成后收到通知，

4.写操作没有错误完成

5.写操作完成时出现错误

Channel 是线程安全(thread-safe)的，它可以被多个不同的线程安全的操作，在多线程环境下，所有的方法都是安全的。正因为 Channel 是安全的，我们存储对Channel的引用，并在学习的时候使用它写入数据到远程已连接的客户端，使用多线程也是如此。下面的代码是一个简单的多线程例子：

Listing 4.6 Using the channel from many threads

```
final Channel channel = ...; // 获取channel的引用
final ByteBuf buf = Unpooled.copiedBuffer("your data",
        CharsetUtil.UTF_8).retain();    //1
Runnable writer = new Runnable() {        //2
    @Override
    public void run() {
        channel.writeAndFlush(buf.duplicate());
    }
};
Executor executor = Executors.newCachedThreadPool();//3

//写进一个线程
executor.execute(writer);        //4

//写进另外一个线程
executor.execute(writer);        //5
```

1.创建一个 ByteBuf 保存写的数据

2.创建 Runnable 用于写数据到 channel

3.获取 Executor 的引用使用线程来执行任务

4.手写一个任务，在一个线程中执行

5.手写另一个任务，在另一个线程中执行

包含的 Transport

Netty 自带了一些传输协议的实现，虽然没有支持所有的传输协议，但是其自带的已足够我们来使用。Netty应用程序的传输协议依赖于底层协议，本节我们将学习Netty中的传输协议。

Netty中的传输方式有如下几种：

Table 4.1 Provided transports

方法名称	包	描述
NIO	io.netty.channel.socket.nio	基于java.nio.channels的工具包，使用选择器作为基础的方法。
OIO	io.netty.channel.socket.oio	基于java.net的工具包，使用阻塞流。
Local	io.netty.channel.local	用来在虚拟机之间本地通信。
Embedded	io.netty.channel.embedded	嵌入传输，它允许在没有真正网络的传输中使用 ChannelHandler，可以非常有用的来测试ChannelHandler的实现。

NIO-Nonblocking I/O

NIO传输是目前最常用的方式，它通过使用选择器提供了完全异步的方式操作所有的 I/O，NIO 从Java 1.4才被提供。

NIO 中，我们可以注册一个通道或获得某个通道的改变的状态，通道状态有下面几种改变：

- 一个新的 Channel 被接受并已准备好
- Channel 连接完成
- Channel 中有数据并已准备好读取
- Channel 发送数据出去

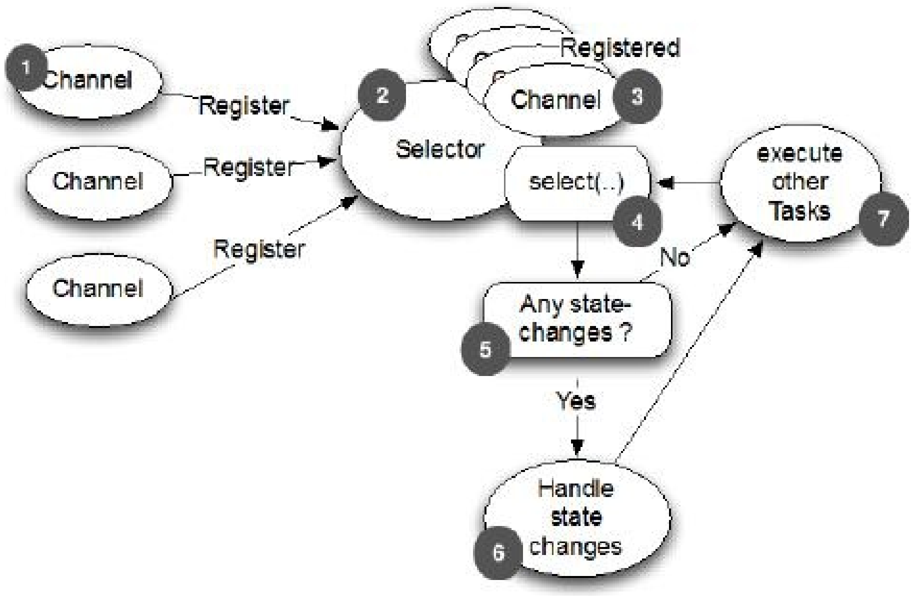
处理完改变的状态后需重新设置他们的状态，用一个线程来检查是否有已准备好的 Channel，如果有则执行相关事件。在这里可能只同时一个注册的事件而忽略其他的。选择器所支持的操作在 SelectionKey 中定义，具体如下：

Table 4.2 Selection operation bit-set

方法名称	描述
OP_ACCEPT	有新连接时得到通知
OP_CONNECT	连接完成后得到通知
OP_REA	准备好读取数据时得到通知
OP_WRITE	写入更多数据到通道时得到通知，大部分时间

这是可能的，但有时 socket 缓冲区完全填满了。这通常发生在你写数据的速度太快了超过了远程节点的处理能力。

Figure 4.2 Selecting and Processing State Changes



1.新信道注册 WITH 选择器

2.选择处理的状态变化的通知

3.以前注册的通道

4.Selector.select（）方法阻塞，直到新的状态变化接收或配置的超时 已过

5.检查是否有状态变化

6.处理所有的状态变化

7.在选择器操作的同一个线程执行其他任务

有一种功能，目前仅适用于 NIO 传输叫什么 “zero-file-copy （零文件拷贝）”，这使您能够快速，高效地通过移动数据到从文件系统传输内容 网络协议栈而无需复制从内核空间到用户空间。这可以使 FT P或 HTTP 协议有很大的不同。

然而，并非所有的操作系统都支持此功能。此外，你不能用它实现数据加密或压缩文件系统 - 仅支持文件的原生内容。另一方面，传送的文件原本已经加密的是完全有效的。

接下来，我们将讨论的是 OIO ，它提供了一个阻塞传输。

OIO-Old blocking I/O

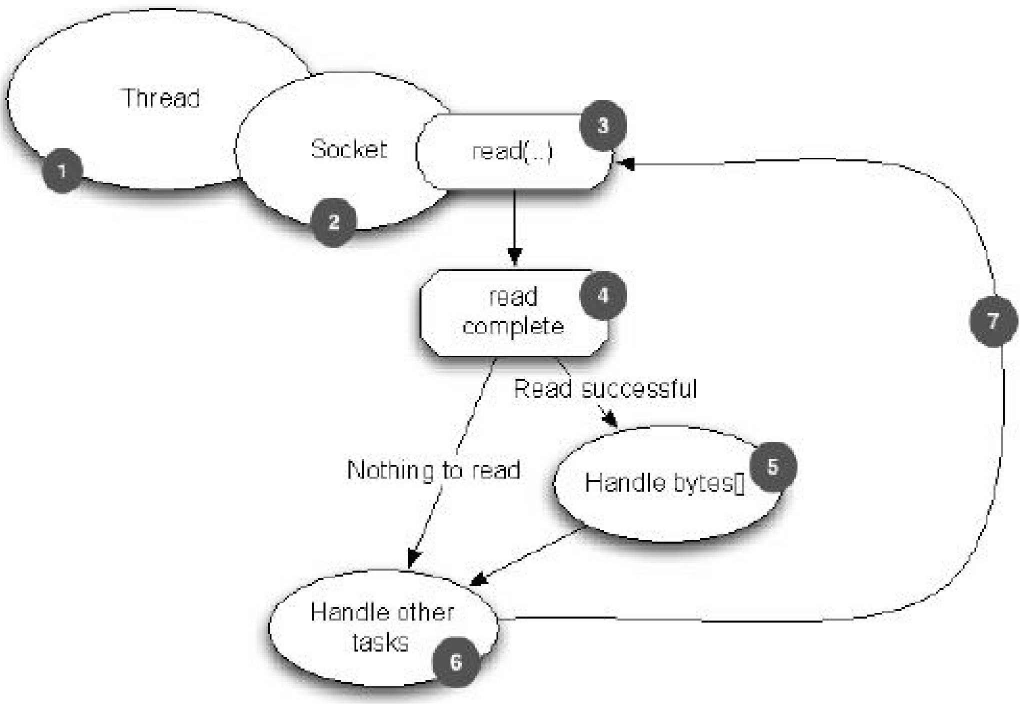
Netty 中，该 OIO 传输代表了一种妥协。它通过了 Netty 的通用 API 访问但不是异步，而是构建在 java.net 的阻塞实现上。任何人下面讨论这一点可能会认为，这个协议并没有很大优势。但它确实有它有效的用途。

假设你需要的端口使用该做阻塞调用库（例如 JDBC）。它可能不适合非阻塞。相反，你可以在短期内使用 OIO 传输，后来移植到纯异步的传输上。让我们看看它是如何工作的。

在 java.net API，你通常有一个线程接受新的连接到达监听在ServerSocket，并创建一个新的线程来处理新的 Socket 。这是必需的，因为在一个特定的 socket的每个 I/O 操作可能会阻塞在任何时间。在一个线程处理多个 socket 易造成阻塞操作，一个 socket 占用了所有的其他人。

鉴于此，你可能想知道 Netty 是如何用相同的 API 来支持 NIO 的异步传输。这里的 Netty 利用了 SO_TIMEOUT 标志，可以设置在一个 Socket。这 timeout 指定最大 毫秒 数量 用于等待 I/O 的操作完成。如果操作在指定的时间内失败，SocketTimeoutException 会被抛出。 Netty中捕获该异常并继续处理循环。在接下来的事件循环运行，它再次尝试。像 Netty 的异步架构来支持 OIO 的话，这其实是唯一的办法。当SocketTimeoutException 抛出时，执行 stack trace。

Figure 4.3 OIO-Processing logic



1.线程分配给 Socket

2.Socket 连接到远程

3.读操作（可能会阻塞）

4.读完成

5.处理可读的字节

6.执行提交到 socket 的其他任务

7.再次尝试读

同个 JVM 内的本地 Transport 通信

Netty 提供了“本地”传输，为运行在同一个 Java 虚拟机上的服务器和客户之间提供异步通信。此传输支持所有的 Netty 常见的传输实现的 API。

在此传输中，与服务器 Channel 关联的 SocketAddress 不是“绑定”到一个物理网络地址中，而是在服务器是运行时它被存储在注册表中，当 Channel 关闭时它会注销。由于该传输不是“真正的”网络通信，它不能与其他传输实现互操作。因此，客户端是希望连接到使用本地传输的的服务器时，要注意正确的用法。除此限制之外，它的使用是与其他的传输是相同的。

内嵌 Transport

Netty中 还提供了可以嵌入 ChannelHandler 实例到其他的 ChannelHandler 的传输，使用它们就像辅助类，增加了灵活性的方法，使您可以与你的 ChannelHandler 互动。

该嵌入技术通常用于测试 ChannelHandler 的实现，但它也可用于将功能添加到现有的 ChannelHandler 而无需更改代码。嵌入传输的关键是Channel 的实现，称为“EmbeddedChannel”。

第10章描述了使用 EmbeddedChannel 来测试 ChannelHandlers。

Transport 使用情况

前面说了，并不是所有传输都支持核心协议，这会限制你的选择，具体看下表

Transport	TCP	UDP	SCTP*	UDT
NIO	X	X	X	X
OIO	X	X	X	X

*指目前仅在 Linux 上的支持。

在 Linux 上启用 SCTP

注意 SCTP 需要 kernel 支持，举例 Ubuntu：

```
sudo apt-get install libsctp1
```

Fedora 使用 yum:

```
sudo yum install kernel-modules-extra.x86_64 lksctp-tools.x86_64
```

虽然只有 SCTP 具有这些特殊的要求，对应的特定的传输也有推荐的配置。想想也是，一个服务器平台可能会需要支持较高的数量的并发连接比单个客户端的话。

下面是你可能遇到的用例:

- OIO-在低连接数、需要低延迟时、阻塞时使用
- NIO-在高连接数时使用
- Local-在同一个JVM内通信时使用
- Embedded-测试ChannelHandler时使用

总结

在本章中，我们研究了传输，他们的实现和使用，以及展示了如何用 Netty来开发。

我们介绍了 Netty 的传输，并解释他们的行为。我们还知道了他们的最低要求，因为不是所有的传输都使用相同的 Java 版本的工作或者可能是仅在特定的操作系统可用。最后，我们讲了匹配传输到特定的用例。

在下一章中，我们的重点是 ByteBuf 和 ByteBufHolder，Netty 中的数据容器。我们将介绍如何使用它们，如何从中获得最佳的性能。

Buffer（缓冲）

正如我们先前所指出的，网络数据的基本单位永远是 byte(字节)。Java NIO 提供 ByteBuffer 作为字节的容器，但它的作用太有限，也没有进行优化。使用ByteBuffer通常是一件繁琐而又复杂的事。

幸运的是，Netty提供了一个强大的缓冲实现类用来表示字节序列以及帮助你操作字节和自定义的POJO。这个新的缓冲类，ByteBuf,效率与JDK的ByteBuffer相当。设计ByteBuf是为了在Netty的pipeline中传输数据。它是为了解决ByteBuffer存在的一些问题以及满足网络程序开发者的需求，以提高他们的生产效率而被设计出来的。

请注意，在本书剩下的章节中，为了帮助区分，我将使用数据容器指代Netty的缓冲接口及实现，同时仍然使用Java的缓冲API指代JDK的缓冲实现。

在本章中，你将会学习Netty的缓冲API,为什么它能够超过JDK的实现，它是如何做到这一点，以及为什么它会比JDK的实现更加灵活。你将会深入了解到如何在Netty框架中访问被交换数据以及你能对它做些什么。这一章是之后章节的基础，因为几乎Netty框架的每一个地方都用到了缓冲。

因为数据需要经过ChannelPipeline和ChannelHandler进行传输，而这又离不开缓冲，所以缓冲在Netty应用程序中是十分普遍的。我们将在第6章学习ChannelHandler和ChannelPipeline。

Buffer API

主要包括

- ByteBuf
- ByteBufHolder

Netty 使用 reference-counting(引用计数)来判断何时可以释放 ByteBuf 或 ByteBufHolder 和其他相关资源，从而可以利用池和其他技巧来提高性能和降低内存的消耗。这一点上不需要开发人员做任何事情，但是在开发 Netty 应用程序时，尤其是使用 ByteBuf 和 ByteBufHolder 时，你应该尽可能早地释放池资源。 Netty 缓冲 API 提供了几个优势：

- 可以自定义缓冲类型
- 通过一个内置的复合缓冲类型实现零拷贝
- 扩展性好，比如 StringBuilder
- 不需要调用 flip() 来切换读/写模式
- 读取和写入索引分开
- 方法链
- 引用计数
- Pooling(池)

ByteBuf - 字节数据的容器

因为所有的网络通信最终都是基于底层的字节流传输，因此一个高效、方便、易用的数据接口是必要的，而 Netty 的 ByteBuf 满足这些需求。

ByteBuf 是一个很好的经过优化的数据容器，我们可以将字节数据有效的添加到 ByteBuf 中或从 ByteBuf 中获取数据。为了便于操作，ByteBuf 提供了两个索引：一个用于读，一个用于写。我们可以按顺序的读取数据，也可以通过调整读取数据的索引或者直接将读取位置索引作为参数传递给get方法来重复读取数据。

ByteBuf 是如何工作的？

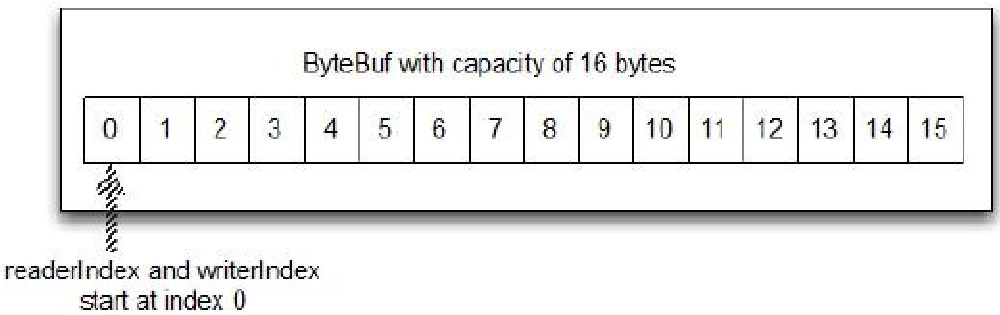
写入数据到 ByteBuf 后，writerIndex（写入索引）增加写入的字节数。读取字节后，readerIndex（读取索引）也增加读取出的字节数。你可以读取字节，直到写入索引和读取索引处在相同的位置。此时ByteBuf不可读，所以下一次读操作将会抛出 IndexOutOfBoundsException，就像读取数组时越位一样。

调用 ByteBuf 的以 "read" 或 "write" 开头的任何方法都将自动增加相应的索引。另一方面，"set" 、 "get"操作字节将不会移动索引位置，它们只会在指定的相对位置上操作字节。

可以给ByteBuf指定一个最大容量值，这个值限制着ByteBuf的容量。任何尝试将写入超过这个值的数据的行为都将导致抛出异常。ByteBuf 的默认最大容量限制是 Integer.MAX_VALUE。

ByteBuf 类似于一个字节数组，最大的区别是读和写的索引可以用来控制对缓冲区数据的访问。下图显示了一个容量为16的空的 ByteBuf 的布局和状态，writerIndex 和 readerIndex 都在索引位置 0 ：

Figure 5.1 A 16-byte ByteBuf with its indices set to 0



ByteBuf 使用模式

HEAP BUFFER(堆缓冲区)

最常用的模式是 ByteBuf 将数据存储在 JVM 的堆空间，这是通过将数据存储在数组的实现。堆缓冲区可以快速分配，当不使用时也可以快速释放。它还提供了直接访问数组的方法，通过 ByteBuf.array() 来获取 byte[]数据。 这种方法，正如清单5.1中所示的那样，是非常适合用来处理遗留数据的。

Listing 5.1 Backing array

```
ByteBuf heapBuf = ...;
if (heapBuf.hasArray()) {                //1
    byte[] array = heapBuf.array();        //2
    int offset = heapBuf.arrayOffset() + heapBuf.readerIndex();                //3
    int length = heapBuf.readableBytes();//4
    handleArray(array, offset, length); //5
}
```

1.检查 ByteBuf 是否有支持数组。

2.如果有的话，得到引用数组。

3.计算第一字节的偏移量。

4.获取可读的字节数。

5.使用数组，偏移量和长度作为调用方法的参数。

注意：

- 访问非堆缓冲区 ByteBuf 的数组会导致UnsupportedOperationException， 可以使用 ByteBuf.hasArray()来检查是否支持访问数组。
- 这个用法与 JDK 的 ByteBuffer 类似

DIRECT BUFFER(直接缓冲区)

“直接缓冲区”是另一个 ByteBuf 模式。对象的所有内存分配发生在 堆，对不对？好吧，并非总是如此。在 JDK1.4 中被引入 NIO 的ByteBuffer 类允许 JVM 通过本地方法调用分配内存，其目的是

- 通过免去中间交换的内存拷贝, 提升IO处理速度; 直接缓冲区的内容可以驻留在垃圾回收扫描的堆区以外。
- DirectBuffer 在 -XX:MaxDirectMemorySize=xxM大小限制下, 使用 Heap 之外的内存, GC对此"无能为力",也就意味着规避了在高负载下频繁的GC过程对应用线程的中断影响.(详见http://docs.oracle.com/javase/7/docs/api/java/nio/ByteBuffer.html.)

这就解释了为什么“直接缓冲区”对于那些通过 socket 实现数据传输的应用来说，是一种非常理想的方式。如果你的数据是存放在堆中分配的缓冲区，那么实际上，在通过 socket 发送数据之前，JVM 需要将先数据复制到直接缓冲区。

但是直接缓冲区的缺点是在内存空间的分配和释放上比堆缓冲区更复杂，另外一个缺点是如果要将数据传递给遗留代码处理，因为数据不是在堆上，你可能不得不作出一个副本，如下：

Listing 5.2 Direct buffer data access

```
ByteBuf directBuf = ...
if (!directBuf.hasArray()) {            //1
    int length = directBuf.readableBytes();//2
    byte[] array = new byte[length];    //3
    directBuf.getBytes(directBuf.readerIndex(), array);        //4
    handleArray(array, 0, length);  //5
}
```

1.检查 ByteBuf 是不是由数组支持。如果不是，这是一个直接缓冲区。

2.获取可读的字节数

3.分配一个新的数组来保存字节

4.字节复制到数组

5.将数组，偏移量和长度作为参数调用某些处理方法

显然，这比使用数组要多做一些工作。因此，如果你事前就知道容器里的数据将作为一个数组被访问，你可能更愿意使用堆内存。

COMPOSITE BUFFER(复合缓冲区)

最后一种模式是复合缓冲区，我们可以创建多个不同的 ByteBuf，然后提供一个这些 ByteBuf 组合的视图。复合缓冲区就像一个列表，我们可以动态的添加和删除其中的 ByteBuf，JDK 的 ByteBuffer 没有这样的功能。

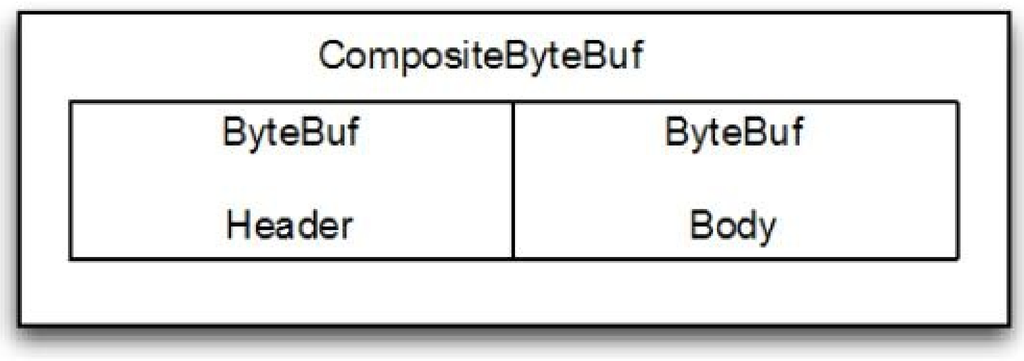
Netty 提供了 ByteBuf 的子类 CompositeByteBuf 类来处理复合缓冲区，CompositeByteBuf 只是一个视图。

警告

CompositeByteBuf.hasArray() 总是返回 *false*，因为它可能既包含堆缓冲区，也包含直接缓冲区

例如，一条消息由 header 和 body 两部分组成，将 header 和 body 组装成一条消息发送出去，可能 body 相同，只是 header 不同，使用CompositeByteBuf 就不用每次都重新分配一个新的缓冲区。下图显示CompositeByteBuf 组成 header 和 body：

Figure 5.2 CompositeByteBuf holding a header and body



下面代码显示了使用 JDK 的 ByteBuffer 的一个实现。两个 ByteBuffer 的数组创建保存消息的组件，第三个创建用于保存所有数据的副本。

Listing 5.3 Composite buffer pattern using ByteBuffer

```
// 使用数组保存消息的各个部分
ByteBuffer[] message = { header, body };

// 使用副本来合并这两个部分
ByteBuffer message2 = ByteBuffer.allocate(
        header.remaining() + body.remaining());
message2.put(header);
message2.put(body);
message2.flip();
```

这种做法显然是低效的;分配和复制操作不是最优的方法，操纵数组使代码显得很笨拙。

下面看使用 CompositeByteBuf 的改进版本

Listing 5.4 Composite buffer pattern using CompositeByteBuf

```
CompositeByteBuf messageBuf = ...;
ByteBuf headerBuf = ...; // 可以支持或直接
ByteBuf bodyBuf = ...; // 可以支持或直接
messageBuf.addComponents(headerBuf, bodyBuf);
// ....
messageBuf.removeComponent(0); // 移除头    //2

for (int i = 0; i < messageBuf.numComponents(); i++) {                        //3
    System.out.println(messageBuf.component(i).toString());
}
```

1.追加 ByteBuf 实例的 CompositeByteBuf

2.删除 索引1的 ByteBuf

3.遍历所有 ByteBuf 实例。

清单5.4 所示，你可以简单地把 CompositeByteBuf 当作一个可迭代遍历的容器。CompositeByteBuf 不允许访问其内部可能存在的支持数组，也不允许直接访问数据，这一点类似于直接缓冲区模式，如图5.5所示。

Listing 5.5 Access data

```
CompositeByteBuf compBuf = ...;
int length = compBuf.readableBytes();    //1
byte[] array = new byte[length];        //2
compBuf.getBytes(compBuf.readerIndex(), array);    //3
handleArray(array, 0, length);    //4
```

1.得到的可读的字节数。

2.分配一个新的数组,数组长度为可读字节长度。

3.读取字节到数组

4.使用数组，把偏移量和长度作为参数

Netty 尝试使用 CompositeByteBuf 优化 socket I/O 操作，消除 原生 JDK 中可能存在的的性能低和内存消耗问题。虽然这是在Netty 的核心代码中进行的优化，并且是不对外暴露的，但是作为开发者还是应该意识到其影响。

CompositeByteBuf API

CompositeByteBuf 提供了大量的附加功能超出了它所继承的 *ByteBuf*。请参阅的 *Netty* 的 *Javadoc* 文档 *API*。

字节级别的操作

除了基本的读写操作， ByteBuf 还提供了它所包含的数据的修改方法。

随机访问索引

ByteBuf 使用zero-based 的 indexing(从0开始的索引)，第一个字节的索引是 0，最后一个字节的索引是 ByteBuf 的 capacity - 1，下面代码是遍历 ByteBuf 的所有字节：

Listing 5.6 Access data

```
ByteBuf buffer = ...;
for (int i = 0; i < buffer.capacity(); i++) {
    byte b = buffer.getByte(i);
    System.out.println((char) b);
}
```

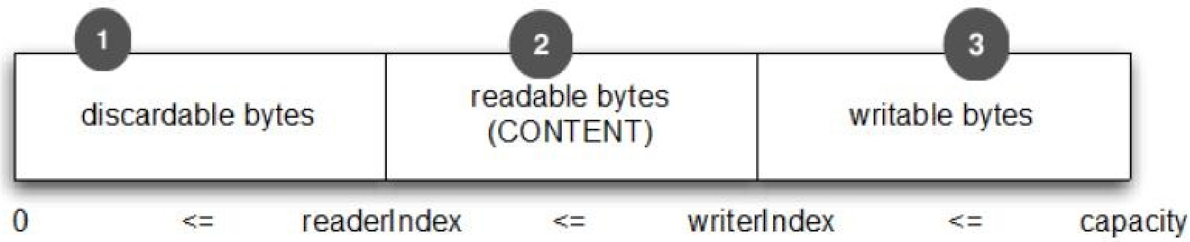
注意通过索引访问时不会推进 readerIndex （读索引）和 writerIndex（写索引），我们可以通过 ByteBuf 的 readerIndex(index) 或 writerIndex(index) 来分别推进读索引或写索引

顺序访问索引

ByteBuf 提供两个指针变量支付读和写操作，读操作是使用 readerIndex()，写操作时使用 writerIndex()。这和JDK的ByteBuffer不同，ByteBuffer只有一个方法来设置索引，所以需要使用 flip() 方法来切换读和写模式。

ByteBuf 一定符合：0 <= readerIndex <= writerIndex <= capacity。

Figure 5.3 ByteBuf internal segmentation



1.字节，可以被丢弃，因为它们已经被读

2.还没有被读的字节是：“readable bytes（可读字节）”

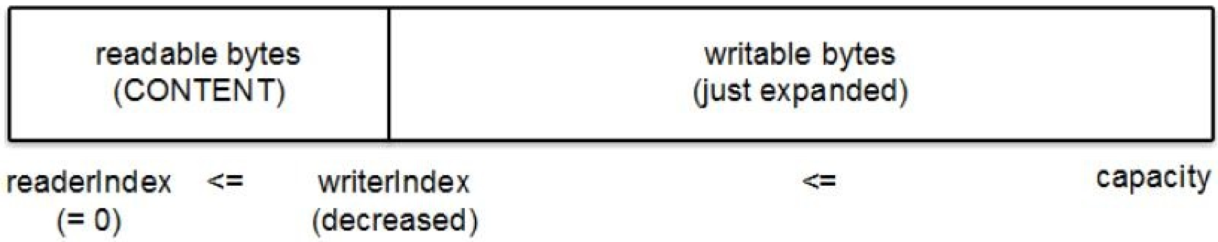
3.空间可加入多个字节的是：“writeable bytes（写字节）”

可丢弃字节的字节

标有“可丢弃字节”的段包含已经被读取的字节。他们可以被丢弃，通过调用 discardReadBytes() 来回收空间。这个段的初始大小存储在readerIndex，为 0，当“read”操作被执行时递增（“get”操作不会移动 readerIndex）。

图5.4示出了在 图5.3 中的缓冲区中调用 discardReadBytes() 所示的结果。你可以看到，在丢弃字节段的空间已变得可用写。需要注意的是不能保证对可写的段之后的内容在 discardReadBytes() 方法之后已经被调用。

Figure 5.4 ByteBuf after discarding read bytes.



1.字节尚未被读出（readerIndex 现在 0）。 2.可用的空间，由于空间被回收而增大。

ByteBuf.discardReadBytes() 可以用来清空 ByteBuf 中已读取的数据，从而使 ByteBuf 有多余的空间容纳新的数据，但是discardReadBytes() 可能会涉及内存复制，因为它需要移动 ByteBuf 中可读的字节到开始位置，这样的操作会影响性能，一般在需要马上释放内存的时候使用收益会比较大。

可读字节

ByteBuf 的“可读字节”分段存储的是实际数据。新分配，包装，或复制的缓冲区的 readerIndex 的默认值为 0 。任何操作，其名称以 "read" 或 "skip" 开头的都将检索或跳过该数据在当前 readerIndex ，并且通过读取的字节数来递增。

如果所谓的读操作是一个指定 ByteBuf 参数作为写入的对象，并且没有一个目标索引参数，目标缓冲区的 writerIndex 也会增加了。例如：

```
readBytes(ByteBuf dest);
```

如果试图从缓冲器读取已经用尽的可读的字节，则抛出IndexOutOfBoundsException。清单 5.8显示了如何读取所有可读字节。

Listing 5.7 Read all data

```
//遍历缓冲区的可读字节
ByteBuf buffer= ...;
while (buffer.isReadable()) {
    System.out.println(buffer.readByte());
}
```

这段是未定义内容的地方，准备好写。一个新分配的缓冲区的 writerIndex 的默认值是 0 。任何操作，其名称 "write"开头的操作在当前的 writerIndex 写入数据时，递增字节写入的数量。如果写操作的目标也是 ByteBuf ，且未指定源索引，则源缓冲区的 readerIndex 将增加相同的量。例如：

```
writeBytes(ByteBuf dest);
```

如果试图写入超出目标的容量，则抛出 IndexOutOfBoundException。

下面的例子展示了填充随机整数到缓冲区中，直到耗尽空间。该方法writableBytes() 被用在这里确定是否存在足够的缓冲空间。

Listing 5.8 Write data

```
//填充随机整数到缓冲区中
ByteBuf buffer = ...;
while (buffer.writableBytes() >= 4) {
    buffer.writeInt(random.nextInt());
}
```

索引管理

在 JDK 的 InputStream 定义了 mark(int readlimit) 和 reset()方法。这些是分别用来标记流中的当前位置和复位流到该位置。

同样，您可以设置和重新定位ByteBuf readerIndex 和 writerIndex 通过调用 markReaderIndex(), markWriterIndex(), resetReaderIndex() 和 resetWriterIndex()。这些类似于InputStream 的调用，所不同的是，没有 readlimit 参数来指定当标志变为无效。

您也可以通过调用 readerIndex(int) 或 writerIndex(int) 将指标移动到指定的位置。在尝试任何无效位置上设置一个索引将导致 IndexOutOfBoundsException 异常。

调用 clear() 可以同时设置 readerIndex 和 writerIndex 为 0。注意，这不会清除内存中的内容。让我们看看它是如何工作的。（图5.5图重复5.3 ）

Figure 5.5 Before clear() is called



调用之前，包含3个段，下面显示了调用之后

Figure 5.6 After clear() is called



现在 整个 ByteBuf 空间都是可写的了。

clear() 比 discardReadBytes() 更低成本，因为他只是重置了索引，而没有内存拷贝。

查询操作

有几种方法，以确定在所述缓冲器中的指定值的索引。最简单的是使用 indexOf() 方法。更复杂的搜索执行以 ByteBufProcessor 为参数的方法。这个接口定义了一个方法，boolean process(byte value)，它用来报告输入值是否是一个正在寻求的值。

ByteBufProcessor 定义了很多方便实现共同目标值。例如，假设您的应用程序需要集成所谓的“Flash sockets”，将使用 NULL 结尾的内容。调用

```
forEachByte（ByteBufProcessor.FIND_NUL）
```

通过减少的，因为少量的 “边界检查”的处理过程中执行了，从而使 消耗 Flash 数据变得 编码工作量更少、效率更高。

下面例子展示了寻找一个回车符，`\ r` 的一个例子。

Listing 5.9 Using ByteBufProcessor to find `\r`

```
ByteBuf buffer = ...;
int index = buffer.forEachByte(ByteBufProcessor.FIND_CR);
```

衍生的缓冲区

“衍生的缓冲区”是代表一个专门的展示 ByteBuf 内容的“视图”。这种视图是由 duplicate(), slice(), slice(int, int),readOnly(), 和 order(ByteOrder) 方法创建的。所有这些都返回一个新的 ByteBuf 实例包括它自己的 reader, writer 和标记索引。然而，内部数据存储共享就像在一个 NIO 的 ByteBuffer。这使得衍生的缓冲区创建、修改其 内容，以及修改其“源”实例更廉价。

ByteBuf 拷贝

如果需要已有的缓冲区的全新副本，使用 *copy()* 或者 *copy(int, int)*。不同于派生缓冲区，这个调用返回的 *ByteBuf* 有数据的独立副本。

若需要操作某段数据，使用 slice(int, int)，下面展示了用法：

Listing 5.10 Slice a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8); //1

ByteBuf sliced = buf.slice(0, 14);          //2
System.out.println(sliced.toString(utf8));  //3

buf.setByte(0, (byte) 'J');                 //4
assert buf.getByte(0) == sliced.getByte(0);
```

1.创建一个 ByteBuf 保存特定字节串。

2.创建从索引 0 开始，并在 14 结束的 ByteBuf 的新 slice。

3.打印 Netty in Action

4.更新索引 0 的字节。

5.断言成功，因为数据是共享的，并以一个地方所做的修改将在其他地方可见。

下面看下如何将一个 ByteBuf 段的副本不同于 slice。

Listing 5.11 Copying a ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);     //1

ByteBuf copy = buf.copy(0, 14);               //2
System.out.println(copy.toString(utf8));      //3

buf.setByte(0, (byte) 'J');                   //4
assert buf.getByte(0) != copy.getByte(0);
```

1.创建一个 ByteBuf 保存特定字节串。

2.创建从索引0开始和 14 结束 的 ByteBuf 的段的拷贝。

3.打印 Netty in Action

4.更新索引 0 的字节。

5.断言成功，因为数据不是共享的，并以一个地方所做的修改将不影响其他。

代码几乎是相同的，但所 衍生的 ByteBuf 效果是不同的。因此，使用一个 slice 可以尽可能避免复制内存。

读/写操作

读/写操作主要由2类：

- gget()/set() 操作从给定的索引开始，保持不变
- read()/write() 操作从给定的索引开始，与字节访问的数量来适用，递增当前的写索引或读索引

ByteBuf 的各种读写方法或其他一些检查方法可以看 ByteBuf 的 API，下面是常见的 get() 操作：

Table 5.1 get() operations

方法名称	描述
getBoolean(int)	返回当前索引的 Boolean 值
getByte(int) getUnsignedByte(int)	返回当前索引的(无符号)字节
getMedium(int) getUnsignedMedium(int)	返回当前索引的 (无符号) 24-bit 中间值
getInt(int) getUnsignedInt(int)	返回当前索引的(无符号) 整型
getLong(int) getUnsignedLong(int)	返回当前索引的 (无符号) Long 型
getShort(int) getUnsignedShort(int)	返回当前索引的 (无符号) Short 型
getBytes(int, ...)	字节

常见 set() 操作如下

Table 5.2 set() operations

方法名称	描述
setBoolean(int, boolean)	在指定的索引位置设置 Boolean 值
setByte(int, int)	在指定的索引位置设置 byte 值
setMedium(int, int)	在指定的索引位置设置 24-bit 中间 值
setInt(int, int)	在指定的索引位置设置 int 值
setLong(int, long)	在指定的索引位置设置 long 值
setShort(int, int)	在指定的索引位置设置 short 值

下面是用法：

Listing 5.12 get() and set() usage

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);    //1
System.out.println((char)buf.getByte(0));                    //2

int readerIndex = buf.readerIndex();                        //3
int writerIndex = buf.writerIndex();

buf.setByte(0, (byte)'B');                            //4

System.out.println((char)buf.getByte(0));                    //5
assert readerIndex == buf.readerIndex();                    //6
assert writerIndex ==  buf.writerIndex();
```

1.创建一个新的 ByteBuf 给指定 String 保存字节

2.打印的第一个字符，`N`

3.存储当前 readerIndex 和 writerIndex

4.更新索引 0 的字符 `B`

5.打印出的第一个字符，现在 `B`

6.这些断言成功，因为这些操作永远不会改变索引

现在，让我们来看看 read() 操作，对当前 readerIndex 或 writerIndex 进行操作。这些用于从 ByteBuf 读取就好像它是一个流。（对应的 write() 操作用于“追加”到 ByteBuf ）。下面展示了常见的 read() 方法。

Table 5.3 read() operations

方法名称	描述
readBoolean()	Reads the Boolean value at the current readerIndex and increases the readerIndex by 1.
readByte() readUnsignedByte()	Reads the (unsigned) byte value at the current readerIndex and increases the readerIndex by 1.
readMedium() readUnsignedMedium()	Reads the (unsigned) 24-bit medium value at the current readerIndex and increases the readerIndex by 3.
readInt() readUnsignedInt()	Reads the (unsigned) int value at the current readerIndex and increases the readerIndex by 4.
readLong() readUnsignedLong()	Reads the (unsigned) int value at the current readerIndex and increases the readerIndex by 8.
readShort() readUnsignedShort()	Reads the (unsigned) int value at the current readerIndex and increases the readerIndex by 2.
readBytes(int,int, ...)	Reads the value on the current readerIndex for the given length into the given object. Also increases the readerIndex by the length.

每个 read() 方法都对应一个 write()。

Table 5.4 Write operations

方法名称	描述
writeBoolean(boolean)	Writes the Boolean value on the current writerIndex and increases the writerIndex by 1.
writeByte(int)	Writes the byte value on the current writerIndex and increases the writerIndex by 1.
writeMedium(int)	Writes the medium value on the current writerIndex and increases the writerIndex by 3.
writeInt(int)	Writes the int value on the current writerIndex and increases the writerIndex by 4.
writeLong(long)	Writes the long value on the current writerIndex and increases the writerIndex by 8.
writeShort(int)	Writes the short value on the current writerIndex and increases thewriterIndex by 2.
writeBytes(int，...）	Transfers the bytes on the current writerIndex from given resources.

Listing 5.13 read()/write() operations on the ByteBuf

```
Charset utf8 = Charset.forName("UTF-8");
ByteBuf buf = Unpooled.copiedBuffer("Netty in Action rocks!", utf8);    //1
System.out.println((char)buf.readByte());                    //2

int readerIndex = buf.readerIndex();                        //3
int writerIndex = buf.writerIndex();                        //4

buf.writeByte((byte)'?');                            //5

assert readerIndex == buf.readerIndex();
assert writerIndex != buf.writerIndex();
```

1.创建一个新的 ByteBuf 保存给定 String 的字节。

2.打印的第一个字符，`N`

3.存储当前的 readerIndex

4.保存当前的 writerIndex

5.更新索引0的字符 `B`

6.此断言成功，因为 writeByte() 在 5 移动了 writerIndex

更多操作

Table 5.5 Other useful operations

方法名称	描述
isReadable()	Returns true if at least one byte can be read.
isWritable()	Returns true if at least one byte can be written.
readableBytes()	Returns the number of bytes that can be read.
writablesBytes()	Returns the number of bytes that can be written.
capacity()	Returns the number of bytes that the ByteBuf can hold. After this it will try to expand again until maxCapacity() is reached.
maxCapacity()	Returns the maximum number of bytes the ByteBuf can hold.
hasArray()	Returns true if the ByteBuf is backed by a byte array.
array()	Returns the byte array if the ByteBuf is backed by a byte array, otherwise throws an

UnsupportedOperationException.

ByteBufHolder

我们经常遇到需要另外存储除有效的实际数据各种属性值。 HTTP 响应是一个很好的例子；与内容一起的字节的还有状态码, cookies,等。

Netty 提供 ByteBufHolder 处理这种常见的情况。 ByteBufHolder 还提供对于 Netty 的高级功能，如缓冲池，其中保存实际数据的 ByteBuf 可以从池中借用，如果需要还可以自动释放。

ByteBufHolder 有那么几个方法。到底层的这些支持接入数据和引用计数。表5.7所示的方法（忽略了那些从继承 ReferenceCounted 的方法）。

Table 5.7 ByteBufHolder operations

名称	描述
data()	返回 ByteBuf 保存的数据
copy()	制作一个 ByteBufHolder 的拷贝，但不共享其数据(所以数据也是拷贝).

如果你想实现一个“消息对象”有效负载存储在 ByteBuf，使用ByteBufHolder 是一个好主意。

ByteBuf 分配

本节介绍 ByteBuf 实例管理的几种方式：

ByteBufAllocator

为了减少分配和释放内存的开销，Netty 通过支持池类 ByteBufAllocator，可用于分配的任何 ByteBuf 我们已经描述过的类型的实例。是否使用池是由应用程序决定的，表5.8列出了 ByteBufAllocator 提供的操作。

Table 5.8 ByteBufAllocator methods

名称	描述
buffer() buffer(int) buffer(int, int)	Return a ByteBuf with heap-based or direct data storage.
heapBuffer() heapBuffer(int) heapBuffer(int, int)	Return a ByteBuf with heap-based storage.
directBuffer() directBuffer(int) directBuffer(int, int)	Return a ByteBuf with direct storage.
compositeBuffer() compositeBuffer(int) heapCompositeBuffer() heapCompositeBuffer(int) directCompositeBuffer()directCompositeBuffer(int)	Return a CompositeByteBuf that can be expanded by adding heapbased or direct buffers.
ioBuffer()	Return a ByteBuf that will be used for I/O operations on a socket.

通过一些方法接受整型参数允许用户指定 ByteBuf 的初始和最大容量值。你可能还记得，ByteBuf 存储可以扩大到其最大容量。

得到一个 ByteBufAllocator 的引用很简单。你可以得到从 Channel （在理论上，每 Channel 可具有不同的 ByteBufAllocator ），或通过绑定到的 ChannelHandler 的 ChannelHandlerContext 得到它，用它实现了你数据处理逻辑。

下面的列表说明获得 ByteBufAllocator 的两种方式。

Listing 5.15 Obtain ByteBufAllocator reference

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc(); //1
....
ChannelHandlerContext ctx = ...;
ByteBufAllocator allocator2 = ctx.alloc(); //2
...
```

1.从 channel 获得 ByteBufAllocator

2.从 ChannelHandlerContext 获得 ByteBufAllocator

Netty 提供了两种 ByteBufAllocator 的实现，一种是 PooledByteBufAllocator,用ByteBuf 实例池改进性能以及内存使用降到最低，此实现使用一个“jemalloc”内存分配。其他的实现不池化 ByteBuf 情况下，每次返回一个新的实例。

Netty 默认使用 PooledByteBufAllocator，我们可以通过 ChannelConfig 或通过引导设置一个不同的实现来改变。更多细节在后面讲述 ，见 Chapter 9, "Bootstrapping Netty Applications"

Unpooled （非池化）缓存

当未引用 ByteBufAllocator 时，上面的方法无法访问到 ByteBuf。对于这个用例 Netty 提供一个实用工具类称为 Unpooled,，它提供了静态辅助方法来创建非池化的 ByteBuf 实例。表5.9列出了最重要的方法

Table 5.9 Unpooled helper class

名称	描述
buffer() buffer(int) buffer(int, int)	Returns an unpooled ByteBuf with heap-based storage
directBuffer() directBuffer(int) directBuffer(int, int)	Returns an unpooled ByteBuf with direct storage
wrappedBuffer()	Returns a ByteBuf, which wraps the given data.
copiedBuffer()	Returns a ByteBuf, which copies the given data

在 非联网项目，该 Unpooled 类也使得它更容易使用的 ByteBuf API，获得一个高性能的可扩展缓冲 API，而不需要 Netty 的其他部分的。

ByteBufUtil

ByteBufUtil 静态辅助方法来操作 ByteBuf，因为这个 API 是通用的，与使用池无关，这些方法已经在外面的分配类实现。

也许最有价值的是 hexDump() 方法，这个方法返回指定 ByteBuf 中可读字节的十六进制字符串，可以用于调试程序时打印 ByteBuf 的内容。一个典型的用途是记录一个 ByteBuf 的内容进行调试。十六进制字符串相比字节而言对用户更友好。 而且十六进制版本可以很容易地转换回实际字节表示。

另一个有用方法是 使用 boolean equals(ByteBuf, ByteBuf),用来比较 ByteBuf 实例是否相等。在 实现自己 ByteBuf 的子类时经常用到。

引用计数器

Netty 4 引入了 引用计数器给 ByteBuf 和 ByteBufHolder（两者都实现了 ReferenceCounted 接口）

引用计数本身并不复杂;它在特定的对象上跟踪引用的数目。实现了ReferenceCounted 的类的实例会通常开始于一个活动的引用计数器为 1。活动的引用计数器大于0的对象被保证不被释放。当数量引用减少到0，该实例将被释放。需要注意的是"释放"的语义是特定于具体的实现。最起码，一个对象，它已被释放应不再可用。

这种技术就是诸如 PooledByteBufAllocator 这种减少内存分配开销的池化的精髓部分。

Listing 5.16 Reference counting

```
Channel channel = ...;
ByteBufAllocator allocator = channel.alloc(); //1
....
ByteBuf buffer = allocator.directBuffer(); //2
assert buffer.refCnt() == 1; //3
...
```

1.从 channel 获取 ByteBufAllocator

2.从 ByteBufAllocator 分配一个 ByteBuf

3.检查引用计数器是否是 1

Listing 5.17 Release reference counted object

```
ByteBuf buffer = ...;
boolean released = buffer.release(); //1
...
```

1.release（）将会递减对象引用的数目。当这个引用计数达到0时，对象已被释放，并且该方法返回 true。

如果尝试访问已经释放的对象，将会抛出 IllegalReferenceCountException 异常。

需要注意的是一个特定的类可以定义自己独特的方式其释放计数的"规则"。 例如，release() 可以将引用计数器直接计为 0 而不管当前引用的对象数目。

谁负责 *release*？

在一般情况下，最后访问的对象负责释放它。在第6章我们会解释 *ChannelHandler* 和 *ChannelPipeline* 的相关概念。

总结

这一章专门讨论了 Netty 基于 ByteBuf 的数据容器。我们开始说明了Netty 比 JDK 更多的优点。我们还突出适合具体情况的 API 的可用变型。

在下一章中，重点是 ChannelHandler，它提供了数据处理逻辑的载体。 ChannelHandler 大量使用了 ByteBuf。

ChannelHandler 和 ChannelPipeline

本章主要内容

- Channel
- ChannelHandler
- ChannePipeline
- ChannelHandlerContext

我们在上一章研究的 ByteBuf 是一个用来“包装”数据的容器。在本章我们将探讨这些容器是如何在应用程序中进行传输的，以及如何处理它们“包装”的数据。

Netty 在这方面提供了强大的支持。它让Channelhandler 链接在ChannelPipeline上使数据处理更加灵活和模块化。

在这一章中，下面我们会遇到各种各样 Channelhandler，ChannelPipeline 的使用案例，以及重要的相关的类Channelhandlercontext 。我们将展示如何将这些基本组成的框架可以帮助我们写干净可重用的处理实现。

ChannelHandler 家族

在我们深入研究 ChannelHandler 内部之前，让我们花几分钟了解下这个 Netty 组件模型的基础。这里先对ChannelHandler 及其子类做个简单的介绍。

Channel 生命周期

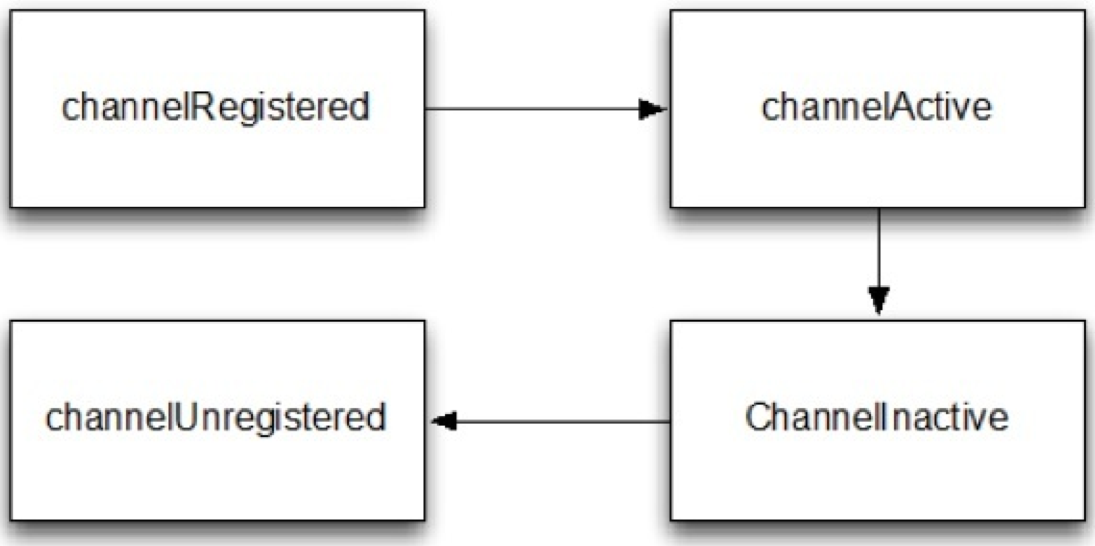
Channel 有个简单但强大的状态模型，与 ChannelInboundHandler API 密切相关。下面表格是 Channel 的四个状态

Table 6.1 Channel lifeycle states

状态	描述
channelUnregistered	channel已创建但未注册到一个 EventLoop.
channelRegistered	channel 注册到一个 EventLoop.
channelActive	channel 变为活跃状态(连接到了远程主机)，现在可以接收和发送数据了
channelInactive	channel 处于非活跃状态，没有连接到远程主机

Channel 的正常的生命周期如下图，当状态出现变化，就会触发对应的事件，这样就能与 ChannelPipeline 中的 ChannelHandler进行及时的交互。

Figure 6.1 Channel State Model



ChannelHandler 生命周期

ChannelHandler 定义的生命周期操作如下表，当 ChannelHandler 添加到 ChannelPipeline，或者从 ChannelPipeline 移除后，对应的方法将会被调用。每个方法都传入了一个 ChannelHandlerContext 参数

Table 6.2 ChannelHandler lifecycle methods

类型	描述
handlerAdded	当 ChannelHandler 添加到 ChannelPipeline 调用
handlerRemoved	当 ChannelHandler 从 ChannelPipeline 移除时调用
exceptionCaught	当 ChannelPipeline 执行抛出异常时调用

ChannelHandler 子接口

Netty 提供2个重要的 ChannelHandler 子接口：

- ChannelInboundHandler - 处理进站数据和所有状态更改事件
- ChannelOutboundHandler - 处理出站数据，允许拦截各种操作

ChannelHandler 适配器

Netty 提供了一个简单的 *ChannelHandler* 框架实现，给所有声明方法签名。这个类 *ChannelHandlerAdapter* 的方法,主要推送事件 到 *pipeline* 下个 *ChannelHandler* 直到 *pipeline* 的结束。这个类 也作为 *ChannelInboundHandlerAdapter* 和 *ChannelOutboundHandlerAdapter* 的基础。所有三个适配器类的目的是作为自己的实现的起点;您可以扩展它们,覆盖你需要自定义的方法。

ChannelInboundHandler

ChannelInboundHandler 的生命周期方法在下表中，当接收到数据或者与之关联的 Channel 状态改变时调用。之前已经注意到了，这些方法与 Channel 的生命周期接近

Table 6.3 ChannelInboundHandler methods

类型	描述
channelRegistered	Invoked when a Channel is registered to its EventLoop and is able to handle I/O.
channelUnregistered	Invoked when a Channel is deregistered from its EventLoop and cannot handle any I/O.
channelActive	Invoked when a Channel is active; the Channel is connected/bound and ready.
channelInactive	Invoked when a Channel leaves active state and is no longer connected to its remote peer.
channelReadComplete	Invoked when a read operation on the Channel has completed.
channelRead	Invoked if data are read from the Channel.
channelWritabilityChanged	Invoked when the writability state of the Channel changes. The user can ensure writes are not done too fast (with risk of an OutOfMemoryError) or can resume writes when the Channel becomes writable again.Channel.isWritable() can be used to detect the actual writability of the channel. The threshold for writability can be set via Channel.config().setWriteHighWaterMark() and Channel.config().setWriteLowWaterMark().
userEventTriggered(...)	Invoked when a user calls Channel.fireUserEventTriggered(...) to pass a pojo through the ChannelPipeline. This can be used to pass user specific events through the ChannelPipeline and so allow handling those events.

注意，ChannelInboundHandler 实现覆盖了 channelRead() 方法处理进来的数据用来响应释放资源。Netty 在 ByteBuf 上使用了资源池，所以当执行释放资源时可以减少内存的消耗。

Listing 6.1 Handler to discard data

```
@ChannelHandler.Sharable
public class DiscardHandler extends ChannelInboundHandlerAdapter {        //1

    @Override
    public void channelRead(ChannelHandlerContext ctx,
                                     Object msg) {
        ReferenceCountUtil.release(msg); //2
    }

}
```

1.扩展 ChannelInboundHandlerAdapter

2.ReferenceCountUtil.release() 来丢弃收到的信息

Netty 用一个 WARN-level 日志条目记录未释放的资源,使其能相当简单地找到代码中的违规实例。然而,由于手工管理资源会很繁琐,您可以通过使用 SimpleChannelInboundHandler 简化问题。如下：

Listing 6.2 Handler to discard data

```
@ChannelHandler.Sharable
public class SimpleDiscardHandler extends SimpleChannelInboundHandler<Object> {  //1

    @Override
    public void channelRead0(ChannelHandlerContext ctx,
                                     Object msg) {
        // No need to do anything special //2
    }

}
```

1.扩展 SimpleChannelInboundHandler

2.不需做特别的释放资源的动作

注意 SimpleChannelInboundHandler 会自动释放资源，而无需存储任何信息的引用。

更多详见 “Error! Reference source not found..” 一节

ChannelOutboundHandler

ChannelOutboundHandler 提供了出站操作时调用的方法。这些方法会被 Channel, ChannelPipeline, 和 ChannelHandlerContext 调用。

ChannelOutboundHandler 另个一个强大的方面是它具有在请求时延迟操作或者事件的能力。比如，当你在写数据到 remote peer 的过程中被意外暂停，你可以延迟执行刷新操作，然后在迟些时候继续。

下面显示了 ChannelOutboundHandler 的方法（继承自 ChannelHandler 未列出来）

Table 6.4 ChannelOutboundHandler methods

类型	描述
bind	Invoked on request to bind the Channel to a local address
connect	Invoked on request to connect the Channel to the remote peer
disconnect	Invoked on request to disconnect the Channel from the remote peer
close	Invoked on request to close the Channel
deregister	Invoked on request to deregister the Channel from its EventLoop
read	Invoked on request to read more data from the Channel
flush	Invoked on request to flush queued data to the remote peer through the Channel
write	Invoked on request to write data through the Channel to the remote peer

几乎所有的方法都将 ChannelPromise 作为参数,一旦请求结束要通过 ChannelPipeline 转发的时候，必须通知此参数。

ChannelPromise vs. ChannelFuture

ChannelPromise 是 特殊的 *ChannelFuture*，允许你的 *ChannelPromise* 及其 操作 成功或失败。所以任何时候调用例如 *Channel.write(...)* 一个新的 *ChannelPromise*将会创建并且通过 *ChannelPipeline*传递。这次写操作本身将会返回 *ChannelFuture*， 这样只允许你得到一次操作完成的通知。*Netty* 本身使用 *ChannelPromise* 作为返回的 *ChannelFuture* 的通知，事实上在大多数时候就是 *ChannelPromise* 自身（*ChannelPromise* 扩展了 *ChannelFuture*）

如前所述,ChannelOutboundHandlerAdapter 提供了一个实现了 ChannelOutboundHandler 所有基本方法的实现的框架。 这些简单事件转发到下一个 ChannelOutboundHandler 管道通过调用 ChannelHandlerContext 相关的等效方法。你可以根据需要自己实现想要的方法。

资源管理

当你通过 ChannelInboundHandler.channelRead(...) 或者 ChannelOutboundHandler.write(...) 来处理数据，重要的是在处理资源时要确保资源不要泄漏。

Netty 使用引用计数器来处理池化的 ByteBuf。所以当 ByteBuf 完全处理后，要确保引用计数器被调整。

引用计数的权衡之一是用户时必须小心使用消息。当 JVM 仍在 GC(不知道有这样的消息引用计数)这个消息，以至于可能是之前获得的这个消息不会被放回池中。因此很可能,如果你不小心释放这些消息，很可能会耗尽资源。

为了让用户更加简单的找到遗漏的释放，Netty 包含了一个 ResourceLeakDetector ，将会从已分配的缓冲区 1% 作为样品来检查是否存在在应用程序泄漏。因为 1% 的抽样,开销很小。

对于检测泄漏,您将看到类似于下面的日志消息。

```
LEAK: ByteBuf.release() was not called before it’s garbage-collected. Enable advanced
leak reporting to find out where the leak occurred. To enable advanced
leak reporting, specify the JVM option ’-Dio.netty.leakDetectionLevel=advanced’ or cal
l ResourceLeakDetector.setLevel()

Relaunch your application with the JVM option mentioned above, then you’ll see the rec
ent locations of your application where the leaked buffer was accessed. The following
output shows a leak from our unit test (XmlFrameDecoderTest.testDecodeWithXml()):

Running io.netty.handler.codec.xml.XmlFrameDecoderTest

15:03:36.886 [main] ERROR io.netty.util.ResourceLeakDetector - LEAK:
ByteBuf.release() was not called before it’s garbage-collected.

Recent access records: 1

#1:

io.netty.buffer.AdvancedLeakAwareByteBuf.toString(AdvancedLeakAwareByteBuf.java:697)

io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithXml(XmlFrameDecoderTest.j
ava:157)
    io.netty.handler.codec.xml.XmlFrameDecoderTest.testDecodeWithTwoMessages(XmlFrameD
ecoderTest.java:133)
```

泄漏检测等级

Netty 现在定义了四种泄漏检测等级，可以按需开启，见下表

Table 6.5 Leak detection levels

Level Description	DISABLED
Disables	Leak detection completely. While this even eliminates the 1 % overhead you should only do this after extensive testing.
SIMPLE	Tells if a leak was found or not. Again uses the sampling rate of 1%, the default level and a good fit for most cases.
ADVANCED	Tells if a leak was found and where the message was accessed, using the sampling rate of 1%.
PARANOID	Same as level ADVANCED with the main difference that every access is sampled. This it has a massive impact on performance. Use this only in the debugging phase.

修改检测等级，只需修改 io.netty.leakDetectionLevel 系统属性，举例

```
# java -Dio.netty.leakDetectionLevel=paranoid
```

这样，我们就能在 ChannelInboundHandler.channelRead(...) 和 ChannelOutboundHandler.write(...) 避免泄漏。

当你处理 channelRead(...) 操作，并在消费消息(不是通过 ChannelHandlerContext.fireChannelRead(...) 来传递它到下个 ChannelInboundHandler) 时，要释放它，如下：

Listing 6.3 Handler that consume inbound data

```
@ChannelHandler.Sharable
public class DiscardInboundHandler extends ChannelInboundHandlerAdapter {  //1

    @Override
    public void channelRead(ChannelHandlerContext ctx,
                                     Object msg) {
        ReferenceCountUtil.release(msg); //2
    }

}
```

1. 继承 ChannelInboundHandlerAdapter
2. 使用 ReferenceCountUtil.release(...) 来释放资源

所以记得，每次处理消息时，都要释放它。

SimpleChannelInboundHandler -消费入站消息更容易

使用入站数据和释放它是一项常见的任务，*Netty* 为你提供了一个特殊的称为 *SimpleChannelInboundHandler* 的 *ChannelInboundHandler* 的实现。该实现将自动释放一个消息，一旦这个消息被用户通过*channelRead0()* 方法消费。

当你在处理写操作，并丢弃消息时，你需要释放它。现在让我们看下实际是如何操作的。

Listing 6.4 Handler to discard outbound data

@ChannelHandler.Sharable public class DiscardOutboundHandler extends ChannelOutboundHandlerAdapter { //1

```
@Override
public void write(ChannelHandlerContext ctx,
                                 Object msg, ChannelPromise promise) {
    ReferenceCountUtil.release(msg);  //2
    promise.setSuccess();    //3

}
```

}

1. 继承 ChannelOutboundHandlerAdapter
2. 使用 ReferenceCountUtil.release(...) 来释放资源
3. 通知 ChannelPromise 数据已经被处理

重要的是，释放资源并通知 ChannelPromise。如果，ChannelPromise 没有被通知到，这可能会引发 ChannelFutureListener 不会被处理的消息通知的状况。

所以，总结下：如果消息是被 消耗/丢弃 并不会被传入下个 ChannelPipeline 的 ChannelOutboundHandler ，调用 ReferenceCountUtil.release(message) 。一旦消息经过实际的传输，在消息被写或者 Channel 关闭时，它将会自动释放。

ChannelPipeline

ChannelPipeline 是一系列的ChannelHandler 实例,用于拦截 流经一个 Channel 的入站和出站事件,ChannelPipeline允许用户自己定义对入站/出站事件的处理逻辑，以及pipeline里的各个Handler之间的交互。

每一次创建了新的Channel ,都会新建一个新的 ChannelPipeline并绑定到Channel上。这个关联是 永久性的;Channel 既不能附上另一个 ChannelPipeline 也不能分离 当前这个。这些都由Netty负责完成,而无需开发人员的特别处理。

根据它的起源,一个事件将由 ChannelInboundHandler 或 ChannelOutboundHandler 处理。随后它将调用 ChannelHandlerContext 实现转发到下一个相同的超类型的处理程序。

ChannelHandlerContext

一个 *ChannelHandlerContext* 使 *ChannelHandler* 与 *ChannelPipeline* 和 其他处理程序交互。一个处理程序可以通知下一个 *ChannelPipeline* 中的 *ChannelHandler* 甚至动态修改 *ChannelPipeline* 的归属。

下图展示了用于入站和出站 ChannelHandler 的 典型 ChannelPipeline 布局。

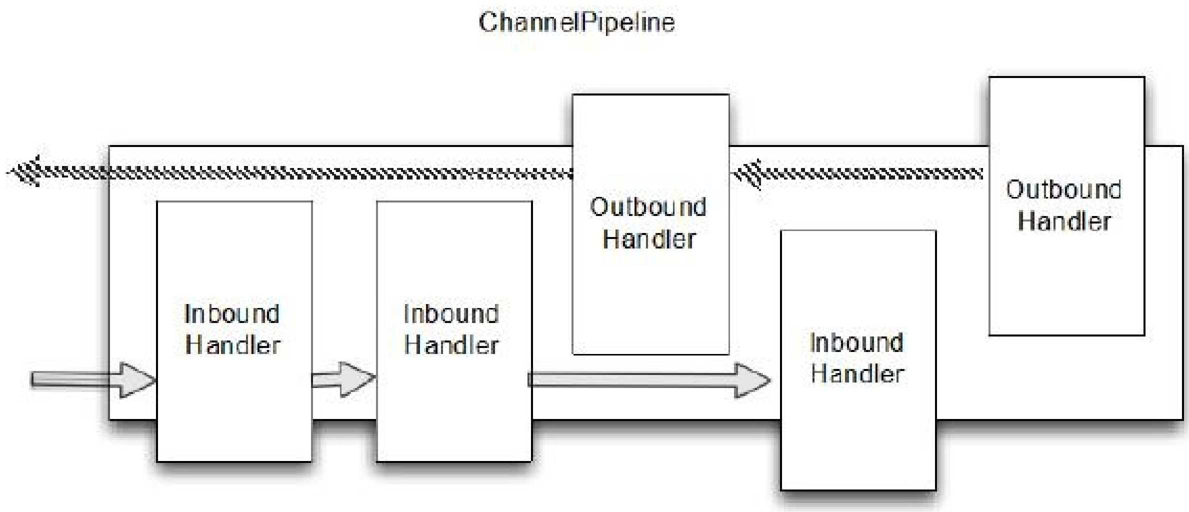


Figure 6.2 ChannelPipeline and ChannelHandlers

上图说明了 ChannelPipeline 主要是一系列 ChannelHandler。通过ChannelPipeline ChannelPipeline 还提供了方法传播事件本身。如果一个入站事件被触发，它将被传递的从 ChannelPipeline 开始到结束。举个例子,在这个图中出站 I/O 事件将从 ChannelPipeline 右端开始一直处理到左边。

ChannelPipeline 相对论

你可能会说,从 *ChannelPipeline* 事件传递的角度来看,*ChannelPipeline* 的“开始”取决于是否入站或出站事件。然而,*Netty* 总是指 *ChannelPipeline* 入站口(图中的左边)为“开始”,出站口(右边)作为“结束”。当我们完成使用 *ChannelPipeline.add*() 添加混合入站和出站处理程序,每个 ChannelHandler 的“顺序”是它的地位从“开始”到“结束”正如我们刚才定义的。因此,如果我们在图6.1处理程序按顺序从左到右第一个ChannelHandler被一个入站事件将是#1,第一个处理程序被出站事件将是#5*

随着管道传播事件,它决定下个 ChannelHandler 是否是相匹配的方向运动的类型。如果没有,ChannelPipeline 跳过 ChannelHandler 并继续下一个合适的方向。记住,一个处理程序可能同时实现ChannelInboundHandler 和 ChannelOutboundHandler 接口。

修改 ChannelPipeline

ChannelHandler 可以实时修改 ChannelPipeline 的布局，通过添加、移除、替换其他 ChannelHandler（也可以从 ChannelPipeline 移除 ChannelHandler 自身）。这个 是 ChannelHandler 重要的功能之一。

Table 6.6 ChannelHandler methods for modifying a ChannelPipeline

名称	描述
addFirst addBefore addAfter addLast	添加 ChannelHandler 到 ChannelPipeline.
Remove	从 ChannelPipeline 移除 ChannelHandler.
Replace	在 ChannelPipeline 替换另外一个 ChannelHandler

下面展示了操作

Listing 6.5 Modify the ChannelPipeline

```
ChannelPipeline pipeline = null; // get reference to pipeline;
FirstHandler firstHandler = new FirstHandler(); //1
pipeline.addLast("handler1", firstHandler); //2
pipeline.addFirst("handler2", new SecondHandler()); //3
pipeline.addLast("handler3", new ThirdHandler()); //4

pipeline.remove("handler3"); //5
pipeline.remove(firstHandler); //6

pipeline.replace("handler2", "handler4", new ForthHandler()); //6
```

1. 创建一个 FirstHandler 实例
2. 添加该实例作为 "handler1" 到 ChannelPipeline
3. 添加 SecondHandler 实例作为 "handler2" 到 ChannelPipeline 的第一个槽，这意味着它

将替换之前已经存在的 "handler1"
4. 添加 ThirdHandler 实例作为"handler3" 到 ChannelPipeline 的最后一个槽
5. 通过名称移除 "handler3"
6. 通过引用移除 FirstHandler (因为只有一个，所以可以不用关联名字 "handler1"）.
7. 将作为"handler2"的 SecondHandler 实例替换为作为 "handler4"的 FourthHandler

以后我们将看到,这种轻松添加、移除和替换 ChannelHandler 能力， 适合非常灵活的实现逻辑。

ChannelHandler 执行 *ChannelPipeline* 和阻塞

通常每个 *ChannelHandler* 添加到 *ChannelPipeline* 将处理事件 传递到 *EventLoop(I/O* 的线程*)*。至关重要的是不要阻塞这个线程， 它将会负面影响的整体处理*I/O*。 有时可能需要使用阻塞 *api* 接口来处理遗留代码。对于这个情况下*,ChannelPipeline* 已有 *add()* 方法*,*它接受一个 *EventExecutorGroup*。如果一个定制的 *EventExecutorGroup* 传入事件将由含在这个 *EventExecutorGroup* 中的 *EventExecutor*之一来处理，并且从 *Channel* 的 *EventLoop* 本身离开。一个默认实现*,*称为来自 *Netty* 的 *DefaultEventExecutorGroup*

除了上述操作，其他访问 ChannelHandler 的方法如下：

Table 6.7 ChannelPipeline operations for retrieving ChannelHandlers

名称	描述
get(...)	Return a ChannelHandler by type or name
context(...)	Return the ChannelHandlerContext bound to a ChannelHandler.
names() iterator()	Return the names or of all the ChannelHander in the ChannelPipeline.

发送事件

ChannelPipeline API 有额外调用入站和出站操作的方法。下表列出了入站操作,用于通知 ChannelPipeline 中 ChannelInboundHandlers 正在发生的事件

Table 6.8 Inbound operations on ChannelPipeline

名称	描述
fireChannelRegistered	Calls channelRegistered(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelUnregistered	Calls channelUnregistered(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelActive	Calls channelActive(ChannelHandlerContext) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelInactive	Calls channelInactive(ChannelHandlerContext)on the next ChannelInboundHandler in the ChannelPipeline.
fireExceptionCaught	Calls exceptionCaught(ChannelHandlerContext, Throwable) on the next ChannelHandler in the ChannelPipeline.
fireUserEventTriggered	Calls userEventTriggered(ChannelHandlerContext, Object) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelRead	Calls channelRead(ChannelHandlerContext, Object msg) on the next ChannelInboundHandler in the ChannelPipeline.
fireChannelReadComplete	Calls channelReadComplete(ChannelHandlerContext) on the next ChannelStateHandler in the ChannelPipeline.

在出站方面,处理一个事件将导致底层套接字的一些行动。下表列出了ChannelPipeline API 出站的操作。

Table 6.9 Outbound operations on ChannelPipeline

名称	描述
bind	Bind the Channel to a local address. This will call bind(ChannelHandlerContext, SocketAddress, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
connect	Connect the Channel to a remote address. This will call connect(ChannelHandlerContext, SocketAddress,ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
disconnect	Disconnect the Channel. This will call disconnect(ChannelHandlerContext, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
close	Close the Channel. This will call close(ChannelHandlerContext,ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
deregister	Deregister the Channel from the previously assigned EventExecutor (the EventLoop). This will call deregister(ChannelHandlerContext,ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline.
flush	Flush all pending writes of the Channel. This will call flush(ChannelHandlerContext) on the next ChannelOutboundHandler in the ChannelPipeline.
write	Write a message to the Channel. This will call write(ChannelHandlerContext, Object msg, ChannelPromise) on the next ChannelOutboundHandler in the ChannelPipeline. Note: this does not write the message to the underlying Socket, but only queues it. To write it to the Socket call flush() or writeAndFlush().
writeAndFlush	Convenience method for calling write() then flush().
read	Requests to read more data from the Channel. This will call read(ChannelHandlerContext) on the next ChannelOutboundHandler in the ChannelPipeline.

总结下：

- 一个 ChannelPipeline 是用来保存关联到一个 Channel 的ChannelHandler
- 可以修改 ChannelPipeline 通过动态添加和删除 ChannelHandler
- ChannelPipeline 有着丰富的API调用动作来回应入站和出站事件。

ChannelHandlerContext

接口 ChannelHandlerContext 代表 ChannelHandler 和ChannelPipeline 之间的关联,并在 ChannelHandler 添加到 ChannelPipeline 时创建一个实例。ChannelHandlerContext 的主要功能是管理通过同一个 ChannelPipeline 关联的 ChannelHandler 之间的交互。

ChannelHandlerContext 有许多方法,其中一些也出现在 Channel 和ChannelPipeline 本身。然而,如果您通过Channel 或ChannelPipeline 的实例来调用这些方法，他们就会在整个pipeline中传播 。相比之下,一样的 的方法在 ChannelHandlerContext的实例上调用， 就只会从当前的 ChannelHandler 开始并传播到相关管道中的下一个有处理事件能力的 ChannelHandler 。

ChannelHandlerContext API 总结如下：

Table 6.10 ChannelHandlerContext API

名称	描述
bind	Request to bind to the given SocketAddress and return a ChannelFuture.
channel	Return the Channel which is bound to this instance.
close	Request to close the Channel and return a ChannelFuture.
connect	Request to connect to the given SocketAddress and return a ChannelFuture.
deregister	Request to deregister from the previously assigned EventExecutor and return a ChannelFuture.
disconnect	Request to disconnect from the remote peer and return a ChannelFuture.
executor	Return the EventExecutor that dispatches events.
fireChannelActive	A Channel is active (connected).
fireChannelInactive	A Channel is inactive (closed).
fireChannelRead	A Channel received a message.
fireChannelReadComplete	Triggers a channelWritabilityChanged event to the next

ChannelInboundHandler. handler | Returns the ChannelHandler bound to this instance. isRemoved | Returns true if the associated ChannelHandler was removed from the ChannelPipeline. name | Returns the unique name of this instance. pipeline | Returns the associated ChannelPipeline. read | Request to read data from the Channel into the first

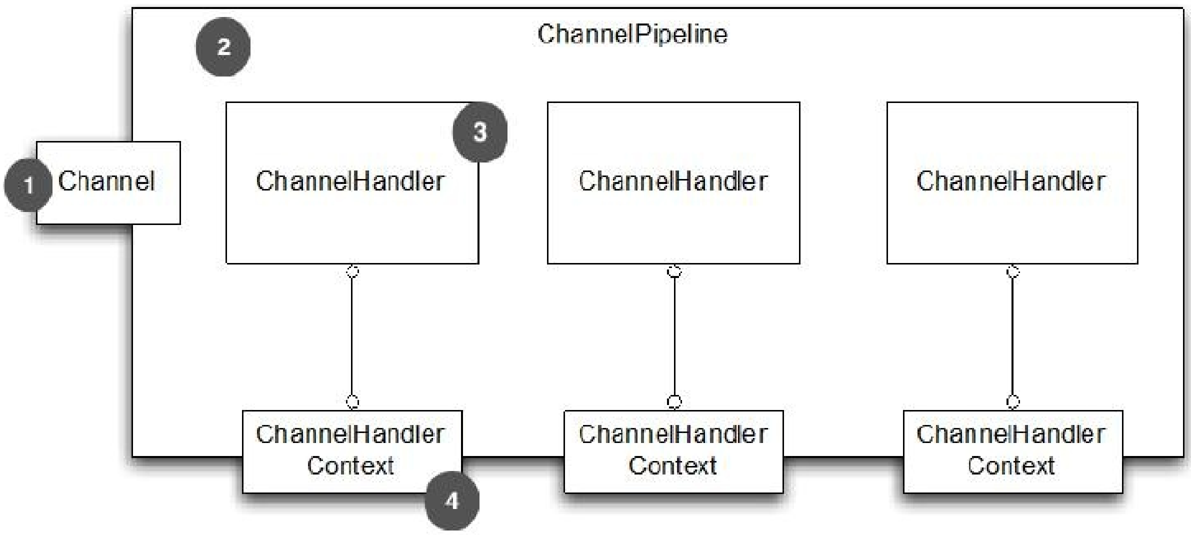
inbound buffer. Triggers a channelRead event if successful and notifies the handler of channelReadComplete. write | Request to write a message via this instance through the pipeline.

其他注意注意事项：

* ChannelHandlerContext 与 ChannelHandler 的关联从不改变，所以缓存它的引用是安全的。
* 正如我们前面指出的,ChannelHandlerContext 所包含的事件流比其他类中同样的方法都要短，利用这一点可以尽可能高地提高性能。

使用 ChannelHandler

本节，我们将说明 ChannelHandlerContext的用法 ，以及ChannelHandlerContext, Channel 和 ChannelPipeline 这些类中方法的不同表现。下图展示了 ChannelPipeline, Channel, ChannelHandler 和 ChannelHandlerContext 的关系



1. Channel 绑定到 ChannelPipeline
2. ChannelPipeline 绑定到 包含 ChannelHandler 的 Channel
3. ChannelHandler
4. 当添加 ChannelHandler 到 ChannelPipeline 时，ChannelHandlerContext 被创建

Figure 6.3 Channel, ChannelPipeline, ChannelHandler and ChannelHandlerContext

下面展示了， 从 ChannelHandlerContext 获取到 Channel 的引用，通过调用 Channel 上的 write() 方法来触发一个 写事件到通过管道的的流中

Listing 6.6 Accessing the Channel from a ChannelHandlerContext

```
ChannelHandlerContext ctx = context;
Channel channel = ctx.channel();  //1
channel.write(Unpooled.copiedBuffer("Netty in Action",
        CharsetUtil.UTF_8));  //2
```

1. 得到与 ChannelHandlerContext 关联的 Channel 的引用
2. 通过 Channel 写缓存

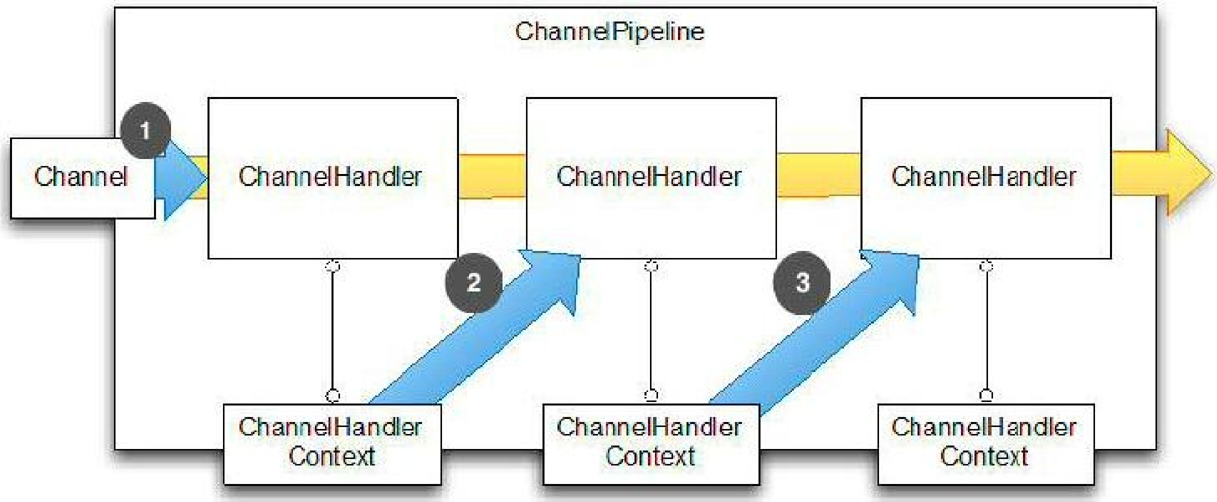
下面展示了 从 ChannelHandlerContext 获取到 ChannelPipeline 的相同示例

Listing 6.7 Accessing the ChannelPipeline from a ChannelHandlerContext

```
ChannelHandlerContext ctx = context;
ChannelPipeline pipeline = ctx.pipeline(); //1
pipeline.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));  //2
```

1. 得到与 ChannelHandlerContext 关联的 ChannelPipeline 的引用
2. 通过 ChannelPipeline 写缓冲区

流在两个清单6.6和6.7是一样的,如图6.4所示。重要的是要注意,虽然在 Channel 或者 ChannelPipeline 上调用write() 都会把事件在整个管道传播,但是在 ChannelHandler 级别上，从一个处理程序转到下一个却要通过在 ChannelHandlerContext 调用方法实现。



1. 事件传递给 ChannelPipeline 的第一个 ChannelHandler
2. ChannelHandler 通过关联的 ChannelHandlerContext 传递事件给 ChannelPipeline 中的下一个
3. ChannelHandler 通过关联的 ChannelHandlerContext 传递事件给 ChannelPipeline 中的下一个

Figure 6.4 Event propagation via the Channel or the ChannelPipeline

为什么你可能会想从 ChannelPipeline 一个特定的点开始传播一个事件?

- 通过减少 ChannelHandler 不感兴趣的事件的传递，从而减少开销
- 排除掉特定的对此事件感兴趣的处理程序的处理

想要实现从一个特定的 ChannelHandler 开始处理，你必须引用与 此ChannelHandler的前一个ChannelHandler 关联的 ChannelHandlerContext 。这个ChannelHandlerContext 将会调用与自身关联的 ChannelHandler 的下一个ChannelHandler 。

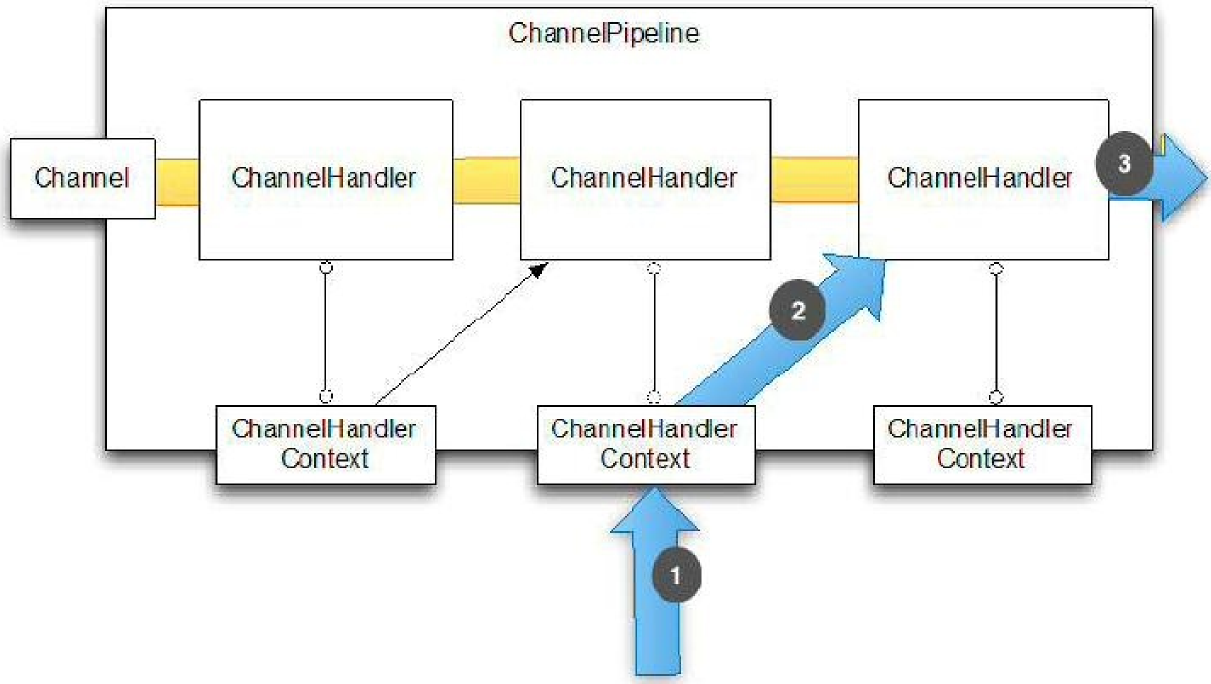
下面展示了使用场景

Listing 6.8 Events via ChannelPipeline

```
ChannelHandlerContext ctx = context;
ctx.write(Unpooled.copiedBuffer("Netty in Action",          CharsetUtil.UTF_8));
```

1. 获得 ChannelHandlerContext 的引用
2. write() 将会把缓冲区发送到下一个 ChannelHandler

如下所示,消息将会从下一个ChannelHandler开始流过 ChannelPipeline ,绕过所有在它之前的ChannelHandler。



1. ChannelHandlerContext 方法调用
2. 事件发送到了下一个 ChannelHandler
3. 经过最后一个ChannelHandler后，事件从 ChannelPipeline 移除

Figure 6.5 Event flow for operations triggered via the ChannelHandlerContext

我们刚刚描述的用例是一种常见的情形,当我们想要调用某个特定的 ChannelHandler操作时，它尤其有用。

ChannelHandler 和 ChannelHandlerContext 的高级用法

正如我们在清单6.6中看到的，通过调用ChannelHandlerContext的 pipeline() 方法，你可以得到一个封闭的 ChannelPipeline 引用。这使得可以在运行时操作 pipeline 的 ChannelHandler，这一点可以被利用来实现一些复杂的需求,例如,添加一个 ChannelHandler 到 pipeline 来支持动态协议改变。

其他高级用例可以实现通过保持一个 ChannelHandlerContext 引用供以后使用,这可能发生在任何 ChannelHandler 方法,甚至来自不同的线程。清单6.9显示了此模式被用来触发一个事件。

Listing 6.9 ChannelHandlerContext usage

```
public class WriteHandler extends ChannelHandlerAdapter {

    private ChannelHandlerContext ctx;

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        this.ctx = ctx;        //1
    }

    public void send(String msg) {
        ctx.writeAndFlush(msg);  //2
    }
}
```

1. 存储 ChannelHandlerContext 的引用供以后使用
2. 使用之前存储的 ChannelHandlerContext 来发送消息

因为 ChannelHandler 可以属于多个 ChannelPipeline ,它可以绑定多个 ChannelHandlerContext 实例。然而,ChannelHandler 用于这种用法必须添加 `@Sharable` 注解。否则,试图将它添加到多个 ChannelPipeline 将引发一个异常。此外,它必须既是线程安全的又能安全地使用多个同时的通道(比如,连接)。

清单6.10显示了此模式的正确实现。

Listing 6.10 A shareable ChannelHandler

```
@ChannelHandler.Sharable            //1
public class SharableHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("channel read message " + msg);
        ctx.fireChannelRead(msg);  //2
    }
}
```

1. 添加 @Sharable 注解
2. 日志方法调用， 并专递到下一个 ChannelHandler

上面这个 ChannelHandler 实现符合所有包含在多个管道的要求;它通过 `@Sharable` 注解，并不持有任何状态。而下面清单6.11中列出的情况则恰恰相反,它会造成问题。

Listing 6.11 Invalid usage of @Sharable

```
@ChannelHandler.Sharable  //1
public class NotSharableHandler extends ChannelInboundHandlerAdapter {
    private int count;

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        count++;  //2

        System.out.println("inboundBufferUpdated(...) called the "
        + count + " time");  //3
        ctx.fireChannelRead(msg);
    }

}
```

1. 添加 @Sharable
2. count 字段递增
3. 日志方法调用， 并专递到下一个 ChannelHandler

这段代码的问题是它持有状态:一个实例变量保持了方法调用的计数。将这个类的一个实例添加到 ChannelPipeline 并发访问通道时很可能产生错误。(当然,这个简单的例子中可以通过在 channelRead() 上添加 synchronized 来纠正)

总之,使用 `@Sharable` 的话，要确定 ChannelHandler 是线程安全的。

为什么共享 *ChannelHandler*

常见原因是要在多个 *ChannelPipelines* 上安装一个 *ChannelHandler* 以此来实现跨多个渠道收集统计数据的目的。

我们的讨论 ChannelHandlerContext 及与其他框架组件关系的 到此结束。接下来我们将解析 Channel 状态模型,准备仔细看看ChannelHandler 本身。

总结

本章带你深入窥探了一下 Netty 的数据处理组件: ChannelHandler。我们讨论了 ChannelHandler 之间是如何链接的以及它在像ChannelInboundHandler 和 ChannelOutboundHandler这样的化身中是如何与 ChannelPipeline 交互的。

下一章将集中在 Netty 的编解码器的抽象上,这种抽象使得编写一个协议编码器和解码器比使用原始 ChannelHandler 接口更容易。

Codec 框架

本章介绍

- Decoder(解码器)
- Encoder(编码器)
- Codec(编解码器)

在前面的章节中,我们讨论了连接到拦截操作或数据处理链的不同方式,展示了如何使用 ChannelHandler 及其相关的类来实现几乎任何一种应用程序所需的逻辑。但正如标准架构模式通常有专门的框架,通用处理模式很适合使用目标实现,可以节省我们大量的开发时间和精力。

在这一章,我们将研究编码和解码——数据从一种特定协议格式到另一种格式的转换。这种处理模式是由通常被称为"codecs(编解码器)"的组件来处理的。Netty提供了一些组件,利用它们可以很容易地为各种不同协议编写编解码器。例如,如果您正在构建一个基于 Netty 的邮件服务器，你可以使用POP3, IMAP 和 SMTP的现成的实现

什么是 Codec

编写一个网络应用程序需要实现某种 codec (编解码器)，codec的作用就是将原始字节数据与目标程序数据格式进行互转。网络中都是以字节码的数据形式来传输数据的，codec 由两部分组成：decoder(解码器)和encoder(编码器)

编码器和解码器一个字节序列转换为另一个业务对象。我们如何区分?

想到一个“消息”是一个结构化的字节序列,语义为一个特定的应用程序——它的“数据”。encoder 是组件,转换消息格式适合传输(就像字节流),而相应的 decoder 转换传输数据回到程序的消息格式。逻辑上,“从”消息转换来是当作操作 outbound（出站）数据,而转换“到”消息是处理 inbound（入站）数据。

我们看看 Netty 的提供的类实现的 codec 。

解码器负责将消息从字节或其他序列形式转成指定的消息对象，编码器则相反；解码器负责处理“入站”数据，编码器负责处理“出站”数据。编码器和解码器的结构很简单，消息被编码后解码后会自动通过ReferenceCountUtil.release(message)释放，如果不想释放消息可以使用ReferenceCountUtil.retain(message)，这将会使引用数量增加而没有消息发布，大多数时候不需要这么做。

Decoder(解码器)

本节，会提供几个类用于 decoder 的实现，并介绍一些具体的例子，这些例子会告诉你什么时候可能用到他们以及怎么来用他们。

Netty 提供了丰富的解码器抽象基类，我们可以很容易的实现这些基类来自定义解码器。主要分两类：

- 解码字节到消息（ByteToMessageDecoder 和 ReplayingDecoder）
- 解码消息到消息（MessageToMessageDecoder）

decoder 负责将“入站”数据从一种格式转换到另一种格式，Netty的解码器是一种 ChannelInboundHandler 的抽象实现。实践中使用解码器很简单，就是将入站数据转换格式后传递到 ChannelPipeline 中的下一个ChannelInboundHandler 进行处理；这样的处理是很灵活的，我们可以将解码器放在 ChannelPipeline 中，重用逻辑。

ByteToMessageDecoder

ByteToMessageDecoder 是用于将字节转为消息（或其他字节序列）。

你不能确定远端是否会一次发送完一个完整的“信息”,因此这个类会缓存入站的数据,直到准备好了用于处理。表7.1说明了它的两个最重要的方法。

Table 7.1 ByteToMessageDecoder API

方法名称	描述
Decode	This is the only abstract method you need to implement. It is called with a ByteBuf having the incoming bytes and a List into which decoded messages are added. decode() is called repeatedly until the List is empty on return. The contents of the List are then passed to the next handler in the pipeline.
decodeLast	The default implementation provided simply calls decode().This method is called once, when the Channel goes inactive. Override to provide special

handling

假设我们接收一个包含简单整数的字节流,每个都单独处理。在本例中,我们将从入站 ByteBuf 读取每个整数并将其传递给 pipeline 中的下一个ChannelInboundHandler。“解码”字节流成整数我们将扩展ByteToMessageDecoder，实现类为“ToIntegerDecoder”,如图7.1所示。

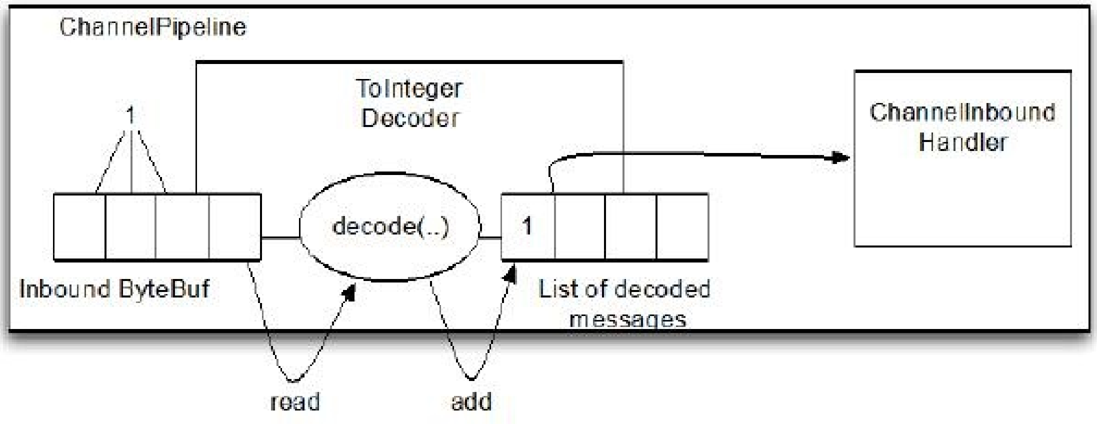


Figure 7.1 ToIntegerDecoder

每次从入站的 ByteBuf 读取四个字节，解码成整形，并添加到一个 List （本例是指 Integer），当不能再添加数据到 list 时，它所包含的内容就会被发送到下个 ChannelInboundHandler

Listing 7.1 ByteToMessageDecoder that decodes to Integer

```
public class ToIntegerDecoder extends ByteToMessageDecoder {  //1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
            throws Exception {
        if (in.readableBytes() >= 4) {  //2
            out.add(in.readInt());  //3
        }
    }
}
```

1. 实现继承了 ByteToMessageDecode 用于将字节解码为消息
2. 检查可读的字节是否至少有4个 (int 是4个字节长度)
3. 从入站 ByteBuf 读取 int ， 添加到解码消息的 List 中

尽管 ByteToMessageDecoder 简化了这个模式,你会发现它还是有点烦人,在实际的读操作(这里 readInt())之前，必须要验证输入的 ByteBuf 要有足够的数据。在下一节中,我们将看看 ReplayingDecoder,一个特殊的解码器。

章节5和6中提到,应该特别注意引用计数。对于编码器和解码器来说，这个过程非常简单。一旦一个消息被编码或解码它自动被调用ReferenceCountUtil.release(message) 。如果你稍后还需要用到这个引用而不是马上释放,你可以调用 ReferenceCountUtil.retain(message)。这将增加引用计数,防止消息被释放。

ReplayingDecoder

ReplayingDecoder 是 byte-to-message 解码的一种特殊的抽象基类，读取缓冲区的数据之前需要检查缓冲区是否有足够的字节，使用ReplayingDecoder就无需自己检查；若ByteBuf中有足够的字节，则会正常读取；若没有足够的字节则会停止解码。

ByteToMessageDecoder 和 ReplayingDecoder

注意到 *ReplayingDecoder* 继承自 *ByteToMessageDecoder* ，所以*API* 跟表 *7.1* 是相同的

也正因为这样的包装使得 ReplayingDecoder 带有一定的局限性：

- 不是所有的标准 ByteBuf 操作都被支持，如果调用一个不支持的操作会抛出 UnreplayableOperationException
- ReplayingDecoder 略慢于 ByteToMessageDecoder

如果这些限制是可以接受你可能更喜欢使用 ReplayingDecoder。下面是一个简单的准则:

如果不引入过多的复杂性 使用 ByteToMessageDecoder 。否则,使用ReplayingDecoder。

Listing 7.2 ReplayingDecoder

```
public class ToIntegerDecoder2 extends ReplayingDecoder<Void> {   //1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
            throws Exception {
        out.add(in.readInt());  //2
    }
}
```

1. 实现继承自 ReplayingDecoder 用于将字节解码为消息
2. 从入站 ByteBuf 读取整型，并添加到解码消息的 List 中

如果我们比较清单7.1和7.2很明显,实现使用 ReplayingDecoder 更简单。

更多 *Decoder*

下面是更加复杂的使用场景： *io.netty.handler.codec.LineBasedFrameDecoder* 通过结束控制符*("\n"* 或 *"\r\n").*解析入站数据。 *io.netty.handler.codec.http.HttpObjectDecoder* 用于 *HTTP* 数据解码

MessageToMessageDecoder

用于从一种消息解码为另外一种消息（例如，POJO 到 POJO），下表展示了方法：

Table 7.2 MessageToMessageDecoder API

方法名称	描述
decode	decode is the only abstract method you need to implement. It is called for each inbound message to be decoded to another format . The decoded messages are then passed to the next ChannelInboundHandler in the pipeline.
decodeLast	The default implementation provided simply calls decode().This method is called once, when the Channel goes inactive. Override to provide special

handling

将 Integer 转为 String，我们提供了 IntegerToStringDecoder，继承自 MessageToMessageDecoder。

因为这是一个参数化的类,实现的签名是:

```
public class IntegerToStringDecoder extends MessageToMessageDecoder<Integer>
```

decode() 方法的签名是

```
protected void decode( ChannelHandlerContext ctx,
Integer msg, List<Object> out ) throws Exception
```

也就是说,入站消息是按照在类定义中声明的参数类型(这里是 Integer) 而不是 ByteBuf来解析的。在之前的例子,解码消息(这里是String)将被添加到List，并传递到下个 ChannelInboundHandler。 这是如图7.2所示。

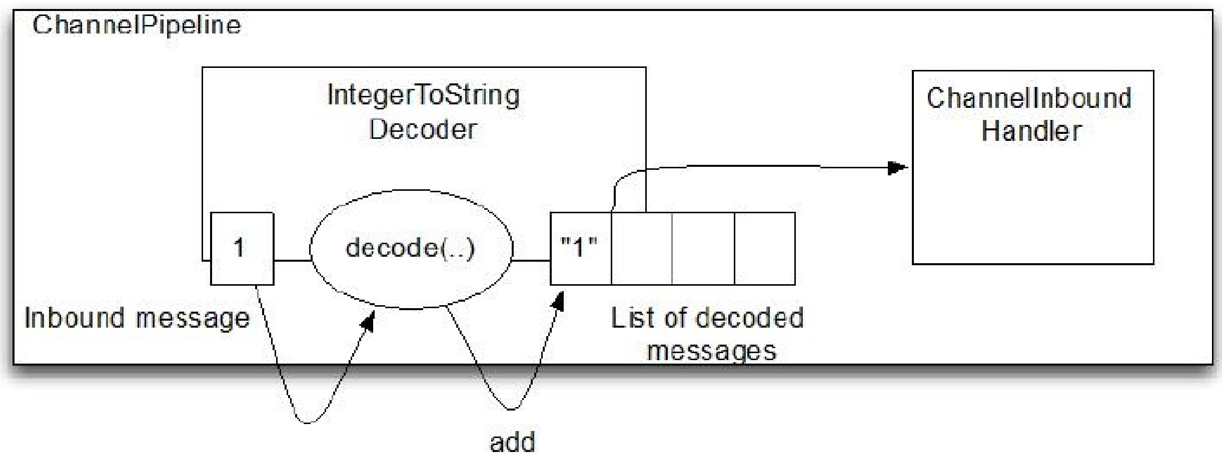


Figure 7.2 IntegerToStringDecoder

实现如下：

Listing 7.3 MessageToMessageDecoder - Integer to String

```
public class IntegerToStringDecoder extends
        MessageToMessageDecoder<Integer> { //1

    @Override
    public void decode(ChannelHandlerContext ctx, Integer msg, List<Object> out)
            throws Exception {
        out.add(String.valueOf(msg)); //2
    }
}
```

1. 实现继承自 MessageToMessageDecoder
2. 通过 String.valueOf() 转换 Integer 消息字符串

正如我们上面指出的,decode()方法的消息参数的类型是由给这个类指定的泛型的类型(这里是Integer)确定的。

HttpObjectAggregator

更多复杂的示例，请查看类 *io.netty.handler.codec.http.HttpObjectAggregator,*继承自 *MessageToMessageDecoder*

在解码时处理太大的帧

Netty 是异步框架需要缓冲区字节在内存中,直到你能够解码它们。因此,你不能让你的解码器缓存太多的数据以免耗尽可用内存。为了解决这个共同关心的问题， Netty 提供了一个 TooLongFrameException ,通常由解码器在帧太长时抛出。

为了避免这个问题,你可以在你的解码器里设置一个最大字节数阈值,如果超出,将导致 TooLongFrameException 抛出(并由 ChannelHandler.exceptionCaught() 捕获)。然后由译码器的用户决定如何处理它。虽然一些协议,比如 HTTP、允许这种情况下有一个特殊的响应,有些可能没有,事件唯一的选择可能就是关闭连接。

如清单7.4所示 ByteToMessageDecoder 可以利用 TooLongFrameException 通知其他 ChannelPipeline 中的 ChannelHandler。

Listing 7.4 SafeByteToMessageDecoder encodes shorts into a ByteBuf

```
public class SafeByteToMessageDecoder extends ByteToMessageDecoder {  //1
    private static final int MAX_FRAME_SIZE = 1024;

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in,
                       List<Object> out) throws Exception {
        int readable = in.readableBytes();
        if (readable > MAX_FRAME_SIZE) { //2
            in.skipBytes(readable);        //3
            throw new TooLongFrameException("Frame too big!");
        }
        // do something
    }
}
```

1. 实现继承 ByteToMessageDecoder 来将字节解码为消息
2. 检测缓冲区数据是否大于 MAX_FRAME_SIZE
3. 忽略所有可读的字节，并抛出 TooLongFrameException 来通知 ChannelPipeline 中的 ChannelHandler 这个帧数据超长

这种保护是很重要的，尤其是当你解码一个有可变帧大小的协议的时候。

到这里我们解释了解码器常见用例和 Netty 提供的用于构建它们的抽象基类。但解码器只是一方面。另一方面,还需要完成 Codec API,我们有编码器,用于转换消息到出站数据。这将是我们下一个话题。

Encoder(编码器)

回顾之前的定义，encoder 是用来把出站数据从一种格式转换到另外一种格式，因此它实现了 ChannelOutboundHandler。正如你所期望的一样，类似于 decoder，Netty 也提供了一组类来帮助你写 encoder，当然这些类提供的是与 decoder 相反的方法，如下所示：

- 编码从消息到字节
- 编码从消息到消息

MessageToByteEncoder

之前我们学习了如何使用 ByteToMessageDecoder 来将字节转换成消息，现在我们使用 MessageToByteEncoder 实现相反的效果。

Table 7.3 MessageToByteEncoder API

方法名称	描述
encode	The encode method is the only abstract method you need to implement. It is called with the outbound message, which this class will encodes to a ByteBuf. The ByteBuf is then forwarded to the next ChannelOutboundHandler in the ChannelPipeline.

这个类只有一个方法，而 decoder 却是有两个，原因就是 decoder 经常需要在 Channel 关闭时产生一个“最后的消息”。出于这个原因，提供了decodeLast()，而 encoder 没有这个需求。

下面示例，我们想产生 Short 值，并想将他们编码成 ByteBuf 来发送到 线上，我们提供了 ShortToByteEncoder 来实现该目的。

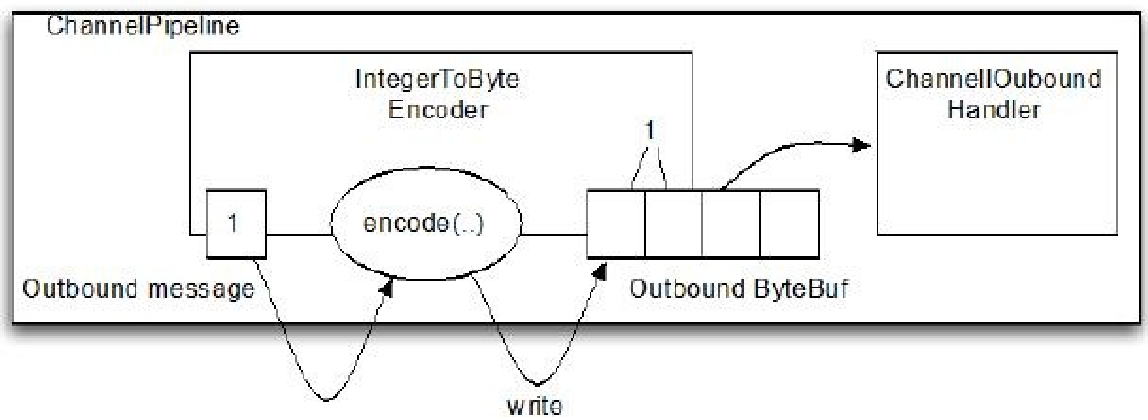


Figure 7.3 ShortToByteEncoder

上图展示了，encoder 收到了 Short 消息，编码他们，并把他们写入 ByteBuf。 ByteBuf 接着前进到下一个 pipeline 的ChannelOutboundHandler。每个 Short 将占用 ByteBuf 的两个字节

Listing 7.5 ShortToByteEncoder encodes shorts into a ByteBuf

```
public class ShortToByteEncoder extends
        MessageToByteEncoder<Short> {  //1
    @Override
    public void encode(ChannelHandlerContext ctx, Short msg, ByteBuf out)
            throws Exception {
        out.writeShort(msg);  //2
    }
}
```

1. 实现继承自 MessageToByteEncoder
2. 写 Short 到 ByteBuf

Netty 提供很多 MessageToByteEncoder 类来帮助你的实现自己的 encoder 。其中 WebSocket08FrameEncoder 就是个不错的范例。可以在 io.netty.handler.codec.http.websocketx 包找到。

MessageToMessageEncoder

我们已经知道了如何将入站数据从一个消息格式解码成另一个格式。现在我们需要一种方法来将出站数据从一种消息编码成另一种消息。MessageToMessageEncoder 提供此功能,见表7.4，同样的只有一个方法,因为不需要产生“最后的消息”。

Table 7.4 MessageToMessageEncoder API

方法名称	描述
encode	The encode method is the only abstract method you need to implement. It is called for each message written with write(...) to encode the message to one or multiple new outbound messages. The encoded messages are then forwarded

下面例子，我们将要解码 Integer 消息到 String 消息。可简单使用 MessageToMessageEncoder

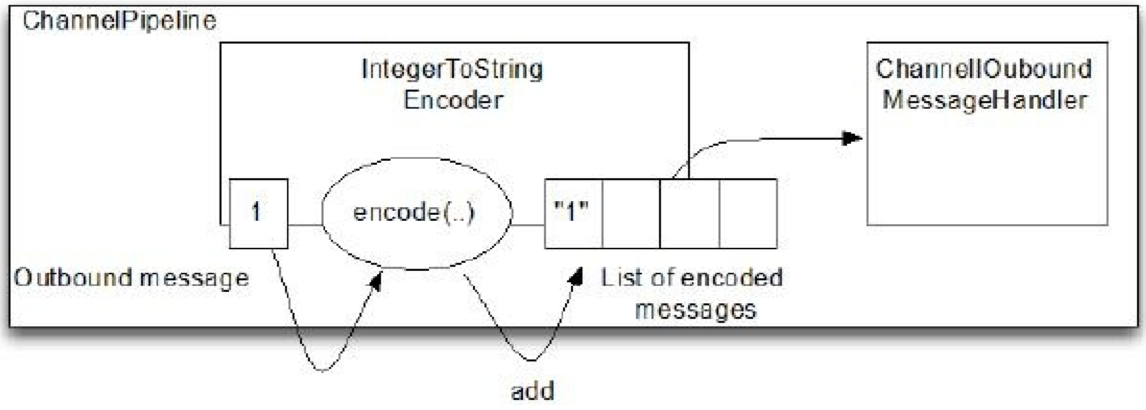


Figure 7.4 IntegerToStringEncoder

encoder 从出站字节流提取 Integer，以 String 形式传递给ChannelPipeline 中的下一个 ChannelOutboundHandler 。清单7.6 显示了细节。

Listing 7.6 IntegerToStringEncoder encodes integer to string

```
public class IntegerToStringEncoder extends
        MessageToMessageEncoder<Integer> { //1

    @Override
    public void encode(ChannelHandlerContext ctx, Integer msg, List<Object> out)
            throws Exception {
        out.add(String.valueOf(msg));  //2
    }
}
```

1. 实现继承自 MessageToMessageEncoder
2. 转 Integer 为 String，并添加到 MessageBuf

更复杂 的 MessageToMessageEncoder 应用案例，可以查看 io.netty.handler.codec.protobuf 包下的 ProtobufEncoder

抽象 Codec(编解码器)类

虽然我们一直把解码器和编码器作为不同的实体来讨论，但你有时可能会发现把入站和出站的数据和信息转换都放在同一个类中更实用。Netty的抽象编解码器类就是用于这个目的,他们把一些成对的解码器和编码器组合在一起，以此来提供对于字节和消息都相同的操作。(这些类实现了 ChannelInboundHandler 和 ChannelOutboundHandler)。

您可能想知道是否有时候使用单独的解码器和编码器会比使用这些组合类要好，最简单的答案是,紧密耦合的两个函数减少了他们的可重用性,但是把他们分开实现就会更容易扩展。当我们研究抽象编解码器类时，我们也会拿它和对应的独立的解码器和编码器做对比。

ByteToMessageCodec

我们需要解码字节到消息,也许是一个 POJO,然后转回来。ByteToMessageCodec 将为我们处理这个问题,因为它结合了ByteToMessageDecoder 和 MessageToByteEncoder。表7.5中列出的重要方法。

Table 7.5 ByteToMessageCodec API

方法名称	描述
decode	This method is called as long as bytes are available to be consumed. It converts the inbound ByteBuf to the specified message format and forwards them to the next ChannelInboundHandler in the pipeline.
decodeLast	The default implementation of this method delegates to decode(). It is called only be called once, when the Channel goes inactive. For special handling it can be oerridden.
encode	This method is called for each message to be written through the ChannelPipeline. The encoded messages are contained in a ByteBuf which

什么会是一个好的 ByteToMessageCodec 用例?任何一个请求/响应协议都可能是,例如 SMTP。编解码器将读取入站字节并解码到一个自定义的消息类型 SmtpRequest 。当接收到一个 SmtpResponse 会产生,用于编码为字节进行传输。

MessageToMessageCodec

7.3.2节中我们看到的一个例子使用 MessageToMessageEncoder 从一个消息格式转换到另一个地方。现在让我们看看 MessageToMessageCodec 是如何处理 单个类 的往返。

在进入细节之前,让我们看看表7.6中的重要方法。

Table 7.6 Methods of MessageToMessageCodec

方法名称	描述
decode	This method is called with the inbound messages of the codec and decodes them to messages. Those messages are forwarded to the next ChannelInboundHandler in the ChannelPipeline
decodeLast	Default implementation delegates to decode().decodeLast will only be called one time, which is when the Channel goes inactive. If you need special handling here you may override decodeLast() to implement it.
encode	The encode method is called for each outbound message to be moved through the ChannelPipeline. The encoded messages are forwarded to the next ChannelOutboundHandler in the pipeline

MessageToMessageCodec 是一个参数化的类，定义如下：

```
public abstract class MessageToMessageCodec<INBOUND,OUTBOUND>
```

上面所示的完整签名的方法都是这样的

```
protected abstract void encode(ChannelHandlerContext ctx,
OUTBOUND msg, List<Object> out)
protected abstract void decode(ChannelHandlerContext ctx,
INBOUND msg, List<Object> out)
```

encode() 处理出站消息类型 OUTBOUND 到 INBOUND，而 decode() 则相反。我们在哪里可能使用这样的编解码器?

在现实中,这是一个相当常见的用例,往往涉及两个来回转换的数据消息传递API 。这是常有的事,当我们不得不与遗留或专有的消息格式进行互操作。

如清单7.7所示这样的可能性。在这个例子中,WebSocketConvertHandler 是一个静态嵌套类，继承了参数为 WebSocketFrame（类型为 INBOUND）和 WebSocketFrame（类型为 OUTBOUND）的 MessageToMessageCode

Listing 7.7 MessageToMessageCodec

```
public class WebSocketConvertHandler extends MessageToMessageCodec<WebSocketFrame, Web
SocketConvertHandler.WebSocketFrame> {  //1

    public static final WebSocketConvertHandler INSTANCE = new WebSocketConvertHandler
();

    @Override
    protected void encode(ChannelHandlerContext ctx, WebSocketFrame msg, List<Object>
out) throws Exception {
        ByteBuf payload = msg.getData().duplicate().retain();
```

```
        switch (msg.getType()) {   //2
            case BINARY:
                out.add(new BinaryWebSocketFrame(payload));
                break;
            case TEXT:
                out.add(new TextWebSocketFrame(payload));
                break;
            case CLOSE:
                out.add(new CloseWebSocketFrame(true, 0, payload));
                break;
            case CONTINUATION:
                out.add(new ContinuationWebSocketFrame(payload));
                break;
            case PONG:
                out.add(new PongWebSocketFrame(payload));
                break;
            case PING:
                out.add(new PingWebSocketFrame(payload));
                break;
            default:
                throw new IllegalStateException("Unsupported websocket msg " + msg);
        }
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, io.netty.handler.codec.http.webso
cketx.WebSocketFrame msg, List<Object> out) throws Exception {
        if (msg instanceof BinaryWebSocketFrame) {  //3
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.BINARY, msg.content().
copy()));
        } else if (msg instanceof CloseWebSocketFrame) {
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.CLOSE, msg.content().c
opy()));
        } else if (msg instanceof PingWebSocketFrame) {
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.PING, msg.content().co
py()));
        } else if (msg instanceof PongWebSocketFrame) {
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.PONG, msg.content().co
py()));
        } else if (msg instanceof TextWebSocketFrame) {
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.TEXT, msg.content().co
py()));
        } else if (msg instanceof ContinuationWebSocketFrame) {
            out.add(new WebSocketFrame(WebSocketFrame.FrameType.CONTINUATION, msg.cont
ent().copy()));
        } else {
            throw new IllegalStateException("Unsupported websocket msg " + msg);
        }
    }

    public static final class WebSocketFrame {  //4
        public enum FrameType {        //5
            BINARY,
```

```
            CLOSE,
            PING,
            PONG,
            TEXT,
            CONTINUATION
        }

        private final FrameType type;
        private final ByteBuf data;
        public WebSocketFrame(FrameType type, ByteBuf data) {
            this.type = type;
            this.data = data;
        }

        public FrameType getType() {
            return type;
        }

        public ByteBuf getData() {
            return data;
        }
    }
}
```

1. 编码 WebSocketFrame 消息转为 WebSocketFrame 消息
2. 检测 WebSocketFrame 的 FrameType 类型，并且创建一个新的响应的 FrameType 类型的 WebSocketFrame
3. 通过 instanceof 来检测正确的 FrameType
4. 自定义消息类型 WebSocketFrame
5. 枚举类明确了 WebSocketFrame 的类型

CombinedChannelDuplexHandler

如前所述,结合解码器和编码器在一起可能会牺牲可重用性。为了避免这种方式，并且部署一个解码器和编码器到 ChannelPipeline 作为逻辑单元而不失便利性。

关键是下面的类:

```
public class CombinedChannelDuplexHandler<I extends ChannelInboundHandler,O extends Ch
annelOutboundHandler>
```

这个类是扩展 ChannelInboundHandler 和 ChannelOutboundHandler 参数化的类型。这提供了一个容器,单独的解码器和编码器类合作而无需直接扩展抽象的编解码器类。我们将在下面的例子说明这一点。首先查看 ByteToCharDecoder ，如清单7.8所示。

Listing 7.8 ByteToCharDecoder

```
public class ByteToCharDecoder extends
        ByteToMessageDecoder { //1

    @Override
    public void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)
            throws Exception {
        if (in.readableBytes() >= 2) {  //2
            out.add(in.readChar());
        }
    }
}
```

1. 继承 ByteToMessageDecoder
2. 写 char 到 MessageBuf

decode() 方法从输入数据中提取两个字节,并将它们作为一个 char 写入 List 。(注意,实现扩展 ByteToMessageDecoder 因为它从 ByteBuf 读取字符。)

现在看一下清单7.9中,把字符转换为字节的编码器。

Listing 7.9 CharToByteEncoder

```
public class CharToByteEncoder extends
        MessageToByteEncoder<Character> { //1

    @Override
    public void encode(ChannelHandlerContext ctx, Character msg, ByteBuf out)
            throws Exception {
        out.writeChar(msg);   //2
    }
}
```

1. 继承 MessageToByteEncoder
2. 写 char 到 ByteBuf

这个实现继承自 MessageToByteEncoder 因为他需要编码 char 消息 到 ByteBuf。这将直接将字符串写为 ByteBuf。

现在我们有编码器和解码器，将他们组成一个编解码器。见下面的 CombinedChannelDuplexHandler.

Listing 7.10 CombinedByteCharCodec

```
public class CombinedByteCharCodec extends CombinedChannelDuplexHandler<ByteToCharDeco
der, CharToByteEncoder> {
    public CombinedByteCharCodec() {
        super(new ByteToCharDecoder(), new CharToByteEncoder());
    }
}
```

1. CombinedByteCharCodec 的参数是解码器和编码器的实现用于处理进站字节和出站消息
2. 传递 ByteToCharDecoder 和 CharToByteEncoder 实例到 super 构造函数来委托调用使他们结合起来。

正如你所看到的,它可能是用上述方式来使程序更简单、更灵活,而不是使用一个以上的编解码器类。它也可以归结到你个人喜好或风格。

总结

在这一章里,我们研究了 Netty 的 codec API 来编写解码器和编码器。我们还学习了为什么最好使用这个而不是纯ChannelHandler API。

我们看到不同的抽象编解码器类提供支持来处理在一个类中实现解码和编码。另一方面,如果我们需要更大的灵活性,希望结合现有实现我们也可以选择结合他们无需扩展抽象编解码器的任何类。

在下一章,我们将讨论 ChannelHandler 的实现和编解码器,他们是 Netty 本身的一部分，您可以开箱即用的处理特定的协议和任务。

已经提供的 ChannelHandler 和 Codec

本章介绍

- 使用 SSL/TLS 加密 Netty 程序
- 构建 Netty HTTP/HTTPS 程序
- 处理空闲连接和超时
- 解码分隔符和基于长度的协议
- 写大数据
- 序列化数据

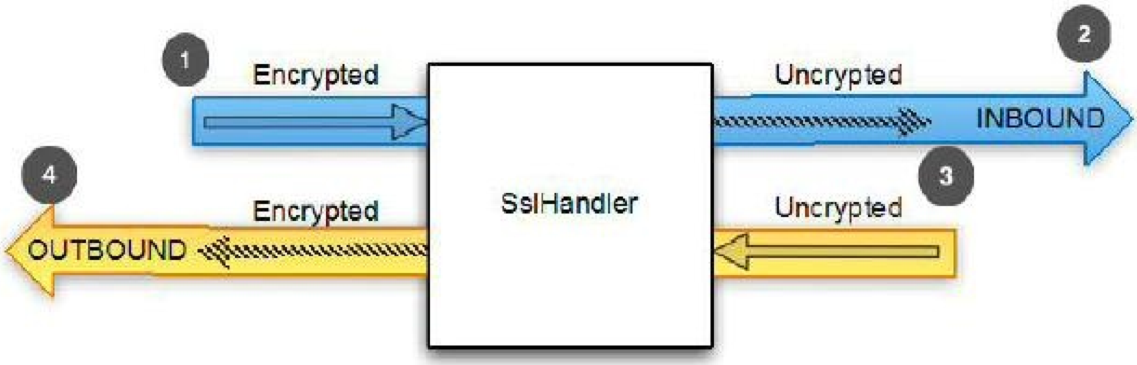
Netty 提供了很多共同协议的编解码器和处理程序,您可以几乎“开箱即用”的使用他们,而无需花在相当乏味的基础设施问题。在这一章里,我们将探索这些工具和他们的好处。这包括支持 SSL/TLS,WebSocket 和 谷歌SPDY,通过数据压缩使 HTTP 有更好的性能。

使用 SSL/TLS 加密 Netty 程序

今天数据隐私是一个十分关注的问题,作为开发人员,我们需要准备好解决这个问题。至少我们需要熟悉加密协议 SSL 和 TLS 等之上的其他协议实现数据安全。作为一个 HTTPS 网站的用户,你是安全。当然,这些协议是广泛不基于 http 的应用程序,例如安全SMTP(SMTPS)邮件服务,甚至关系数据库系统。

为了支持 SSL/TLS,Java 提供了 javax.net.ssl API 的类SslContext 和 SslEngine 使它相对简单的实现解密和加密。Netty 的利用该 API 命名 SslHandler 的 ChannelHandler 实现, 有一个内部 SslEngine 做实际的工作。

图8.1显示了一个使用 SslHandler 数据流图。



1. 加密的入站数据被 SslHandler 拦截，并被解密
2. 前面加密的数据被 SslHandler 解密
3. 平常数据传过 SslHandler
4. SslHandler 加密数据并它传递出站

Figure 8.1 Data flow through SslHandler for decryption and encryption

如清单8.1所示一个 SslHandler 使用 ChannelInitializer 添加到 ChannelPipeline。(回想一下,当 Channel 注册时 ChannelInitializer 用于设置 ChannelPipeline 。)

Listing 8.1 Add SSL/TLS support

```
public class SslChannelInitializer extends ChannelInitializer<Channel> {

    private final SslContext context;
    private final boolean startTls;
    public SslChannelInitializer(SslContext context,
    boolean client, boolean startTls) {   //1
        this.context = context;
        this.startTls = startTls;
    }
    @Override
    protected void initChannel(Channel ch) throws Exception {
        SSLEngine engine = context.newEngine(ch.alloc());  //2
        engine.setUseClientMode(client); //3
        ch.pipeline().addFirst("ssl", new SslHandler(engine, startTls));  //4
    }
}
```

1. 使用构造函数来传递 SSLContext 用于使用(startTls 是否启用)
2. 从 SslContext 获得一个新的 SslEngine 。给每个 SslHandler 实例使用一个新的 SslEngine
3. 设置 SslEngine 是 client 或者是 server 模式
4. 添加 SslHandler 到 pipeline 作为第一个处理器

在大多数情况下,SslHandler 将成为 ChannelPipeline 中的第一个 ChannelHandler 。这将确保所有其他 ChannelHandler 应用他们的逻辑到数据后加密后才发生,从而确保他们的变化是安全的。

SslHandler 有很多有用的方法,如表8.1所示。例如,在握手阶段两端相互验证,商定一个加密方法。您可以配置 SslHandler 修改其行为或提供 在SSL/TLS 握手完成后发送通知,这样所有数据都将被加密。 SSL/TLS 握手将自动执行。

Table 8.1 SslHandler methods

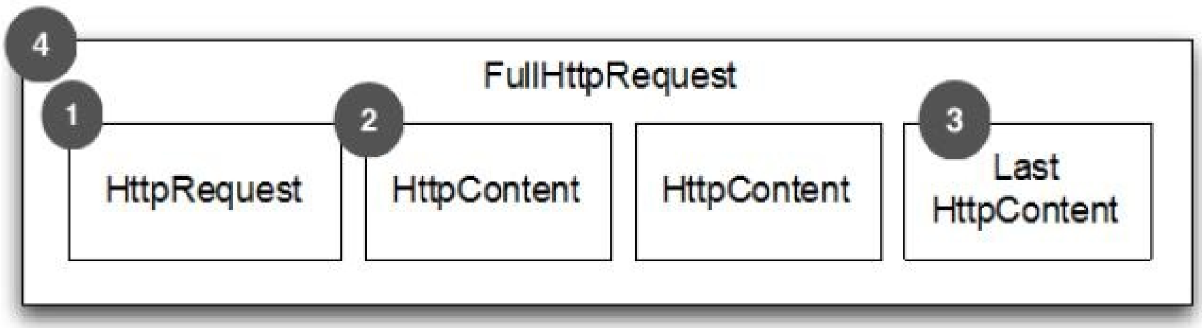
名称	描述
setHandshakeTimeout(...) setHandshakeTimeoutMillis(...) getHandshakeTimeoutMillis()	Set and get the timeout, after which the handshake ChannelFuture is notified of failure.
setCloseNotifyTimeout(...) setCloseNotifyTimeoutMillis(...) getCloseNotifyTimeoutMillis()	Set and get the timeout after which the close notify will time out and the connection will close. This also results in having the close notify ChannelFuture fail.
handshakeFuture()	Returns a ChannelFuture that will be notified once the handshake is complete. If the handshake was done before it will return a ChannelFuture that contains the result of the previous handshake.
close(...)	Send the close_notify to request close and destroy the underlying SslEngine.

构建 Netty HTTP/HTTPS 应用

HTTP/HTTPS 是最常见的一种协议，在智能手机里广泛应用。虽然每家公司都有一个主页,您可以通过HTTP或HTTPS访问,这不是它唯一的使用。许多组织通过 HTTP(S) 公开 WebService API ,旨在用于缓解独立的平台带来的弊端 。 让我们看一下 Netty 提供的 ChannelHandler,是如何允许您使用 HTTP 和 HTTPS 而无需编写自己的编解码器。

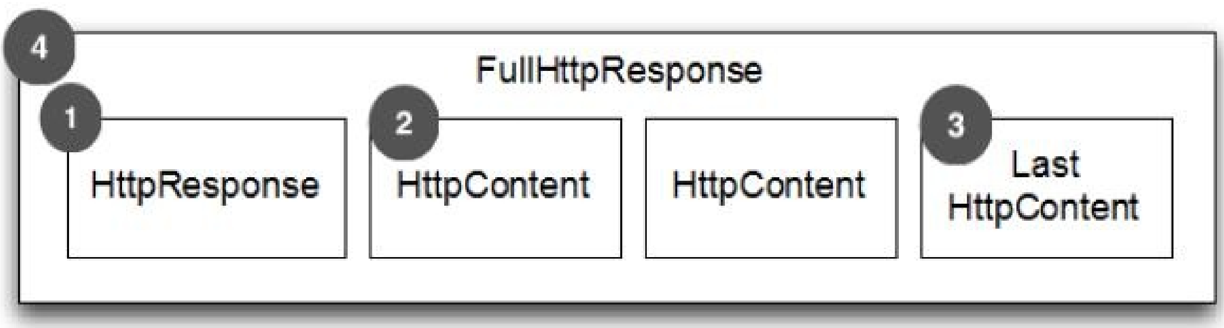
HTTP Decoder, Encoder 和 Codec

HTTP 是请求-响应模式，客户端发送一个 HTTP 请求，服务就响应此请求。Netty 提供了简单的编码、解码器来简化基于这个协议的开发工作。图8.2和图8.3显示 HTTP 请求和响应的方法是如何生产和消费的



1. HTTP Request 第一部分是包含的头信息
2. HttpContent 里面包含的是数据，可以后续有多个 HttpContent 部分
3. LastHttpContent 标记是 HTTP request 的结束，同时可能包含头的尾部信息
4. 完整的 HTTP request

Figure 8.2 HTTP request component parts



1. HTTP response 第一部分是包含的头信息
2. HttpContent 里面包含的是数据，可以后续有多个 HttpContent 部分
3. LastHttpContent 标记是 HTTP response 的结束，同时可能包含头的尾部信息
4. 完整的 HTTP response

Figure 8.3 HTTP response component parts

如图8.2和8.3所示的 HTTP 请求/响应可能包含不止一个数据部分,它总是终止于 LastHttpContent 部分。FullHttpRequest 和FullHttpResponse 消息是特殊子类型,分别表示一个完整的请求和响应。所有类型的 HTTP 消息(FullHttpRequest ，LastHttpContent 以及那些如清单8.2所示)实现 HttpObject 接口。

表8.2概述 HTTP 解码器和编码器的处理和生产这些消息。

Table 8.2 HTTP decoder and encoder

名称	描述
HttpRequestEncoder	Encodes HttpRequest , HttpContent and LastHttpContent messages to bytes.
HttpResponseEncoder	Encodes HttpResponse, HttpContent and LastHttpContent messages to bytes.
HttpRequestDecoder	Decodes bytes into HttpRequest, HttpContent and LastHttpContent messages.
HttpResponseDecoder	Decodes bytes into HttpResponse, HttpContent and LastHttpContent messages.

清单8.2所示的是将支持 HTTP 添加到您的应用程序是多么简单。仅仅添加正确的 ChannelHandler 到 ChannelPipeline 中

Listing 8.2 Add support for HTTP

```
public class HttpPipelineInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpPipelineInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("decoder", new HttpResponseDecoder());  //1
            pipeline.addLast("encoder", new HttpRequestEncoder());  //2
        } else {
            pipeline.addLast("decoder", new HttpRequestDecoder());  //3
            pipeline.addLast("encoder", new HttpResponseEncoder());  //4
        }
    }
}
```

1. client: 添加 HttpResponseDecoder 用于处理来自 server 响应
2. client: 添加 HttpRequestEncoder 用于发送请求到 server
3. server: 添加 HttpRequestDecoder 用于接收来自 client 的请求
4. server: 添加 HttpResponseEncoder 用来发送响应给 client

HTTP消息聚合

安装 ChannelPipeline 中的初始化之后,你能够对不同 HttpObject 消息进行操作。但由于 HTTP 请求和响应可以由许多部分组合而成，你需要聚合他们形成完整的消息。为了消除这种繁琐任务， Netty 提供了一个聚合器,合并消息部件到 FullHttpRequest 和 FullHttpResponse 消息。这样您总是能够看到完整的消息内容。

这个操作有一个轻微的成本,消息段需要缓冲,直到完全可以将消息转发到下一个 ChannelInboundHandler 管道。但好处是,你不必担心消息碎片。

实现自动聚合只需添加另一个 ChannelHandler 到 ChannelPipeline。清单8.3显示了这是如何实现的。

Listing 8.3 Automatically aggregate HTTP message fragments

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean client;

    public HttpAggregatorInitializer(boolean client) {
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());  //1
        } else {
            pipeline.addLast("codec", new HttpServerCodec());  //2
        }
        pipeline.addLast("aggegator", new HttpObjectAggregator(512 * 1024));  //3
    }
}
```

1. client: 添加 HttpClientCodec
2. server: 添加 HttpServerCodec 作为我们是 server 模式时
3. 添加 HttpObjectAggregator 到 ChannelPipeline, 使用最大消息值是 512kb

HTTP 压缩

使用 HTTP 时建议压缩数据以减少传输流量，压缩数据会增加 CPU 负载，现在的硬件设施都很强大，大多数时候压缩数据时一个好主意。Netty 支持“gzip”和“deflate”，为此提供了两个 ChannelHandler 实现分别用于压缩和解压。看下面代码：

HTTP Request Header

客户端可以通过提供下面的头显示支持加密模式。然而服务器不是,所以不得不压缩它发送的数据。

```
GET /encrypted-area HTTP/1.1
Host: www.example.com
Accept-Encoding: gzip, deflate
```

下面是一个例子

Listing 8.4 Automatically compress HTTP messages

```
public class HttpAggregatorInitializer extends ChannelInitializer<Channel> {

    private final boolean isClient;
    public HttpAggregatorInitializer(boolean isClient) {
        this.isClient = isClient;
    }
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        if (isClient) {
            pipeline.addLast("codec", new HttpClientCodec()); //1
            pipeline.addLast("decompressor",new HttpContentDecompressor()); //2
        } else {
            pipeline.addLast("codec", new HttpServerCodec()); //3
            pipeline.addLast("compressor",new HttpContentCompressor()); //4
        }
    }
}
```

1. client: 添加 HttpClientCodec
2. client: 添加 HttpContentDecompressor 用于处理来自服务器的压缩的内容
3. server: HttpServerCodec
4. server: HttpContentCompressor 用于压缩来自 client 支持的 HttpContentCompressor

压缩与依赖

注意，*Java 6*或者更早版本，如果要压缩数据，需要添加 *jzlib* 到 *classpath*

```
<dependency>
    <groupId>com.jcraft</groupId>
        <artifactId>jzlib</artifactId>
    <version>1.1.3</version>
</dependency>
```

使用 HTTPS

启用 HTTPS，只需添加 SslHandler

Listing 8.5 Using HTTPS

```
public class HttpsCodecInitializer extends ChannelInitializer<Channel> {

    private final SslContext context;
    private final boolean client;

    public HttpsCodecInitializer(SslContext context, boolean client) {
        this.context = context;
        this.client = client;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.newEngine(ch.alloc());
        pipeline.addFirst("ssl", new SslHandler(engine));  //1

        if (client) {
            pipeline.addLast("codec", new HttpClientCodec());  //2
        } else {
            pipeline.addLast("codec", new HttpServerCodec());  //3
        }
    }
}
```

1. 添加 SslHandler 到 pipeline 来启用 HTTPS
2. client: 添加 HttpClientCodec
3. server: 添加 HttpServerCodec ，如果是 server 模式的话

上面的代码就是一个很好的例子，解释了 Netty 的架构是如何让“重用”变成了“杠杆”。我们可以添加一个新的功能,甚至是一样重要的加密支持,几乎没有工作量,只需添加一个 ChannelHandler 到 ChannelPipeline。

WebSocket

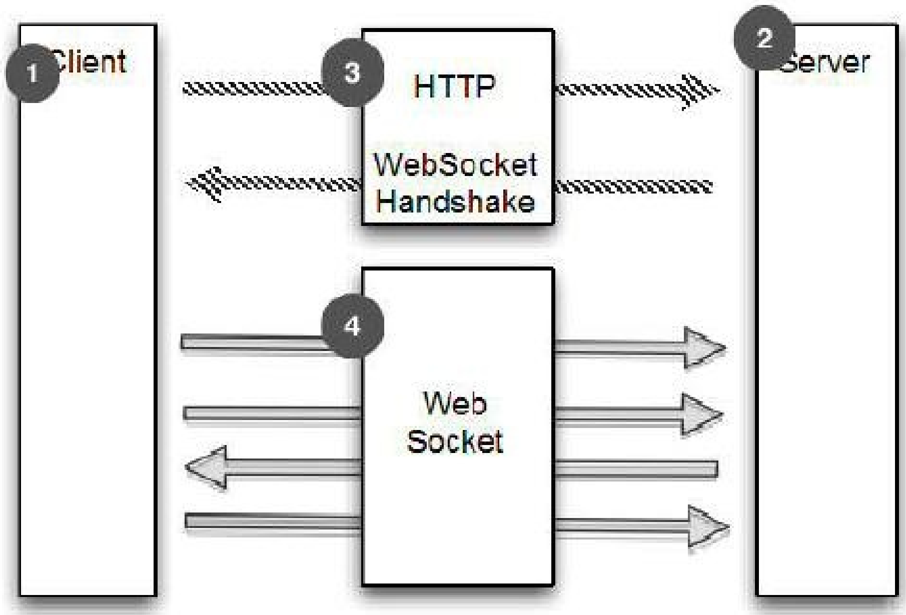
HTTP 是不错的协议，但是如果需要实时发布信息怎么做？有个做法就是客户端一直轮询请求服务器，这种方式虽然可以达到目的，但是其缺点很多，也不是优秀的解决方案，为了解决这个问题，便出现了 WebSocket。

WebSocket 允许数据双向传输，而不需要请求-响应模式。早期的WebSocket 只能发送文本数据，然后现在不仅可以发送文本数据，也可以发送二进制数据，这使得可以使用 WebSocket 构建你想要的程序。下图是WebSocket 的通信示例图：

WebSocket 规范及其实现是为了一个更有效的解决方案。简单的说, 一个WebSocket 提供一个 TCP 连接两个方向的交通。结合 WebSocket API 它提供了一个替代 HTTP 轮询双向通信从页面到远程服务器。

也就是说,WebSocket 提供真正的双向客户机和服务器之间的数据交换。 我们不会对内部太多的细节,但我们应该提到,虽然最早实现仅限于文本数据，但现在不再是这样,WebSocket可以用于任意数据,就像一个正常的套接字。

图8.4给出了一个通用的 WebSocket 协议。在这种情况下的通信开始于普通 HTTP ，并“升级”为双向 WebSocket。



1. Client (HTTP) 与 Server 通讯
2. Server (HTTP) 与 Client 通讯
3. Client 通过 HTTP(s) 来进行 WebSocket 握手,并等待确认
4. 连接协议升级至 WebSocket

Figure 8.4 WebSocket protocol

添加应用程序支持 WebSocket 只需要添加适当的客户端或服务器端WebSocket ChannelHandler 到管道。这个类将处理特殊 WebSocket 定义的消息类型,称为“帧。“如表8.3所示,这些可以归类为“数据”和“控制”帧。

Table 8.3 WebSocketFrame types

名称	描述
BinaryWebSocketFrame	Data frame: binary data
TextWebSocketFrame	Data frame: text data
ContinuationWebSocketFrame	Data frame: text or binary data that belongs to a previous BinaryWebSocketFrame or TextWebSocketFrame
CloseWebSocketFrame	Control frame: a CLOSE request, close status code and a phrase
PingWebSocketFrame	Control frame: requests the send of a PongWebSocketFrame
PongWebSocketFrame	Control frame: sent as response to a PingWebSocketFrame

由于 Netty 的主要是一个服务器端技术重点在这里创建一个 WebSocket server 。清单8.6使用 WebSocketServerProtocolHandler 提出了一个简单的例子。该类处理协议升级握手以及三个“控制”帧 Close, Ping 和 Pong。Text 和 Binary 数据帧将被传递到下一个处理程序(由你实现)进行处理。

Listing 8.6 Support WebSocket on the server

```
public class WebSocketServerInitializer extends ChannelInitializer<Channel> {
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(
                new HttpServerCodec(),
                new HttpObjectAggregator(65536),  //1
                new WebSocketServerProtocolHandler("/websocket"),  //2
                new TextFrameHandler(),  //3
                new BinaryFrameHandler(),  //4
                new ContinuationFrameHandler());  //5
    }

    public static final class TextFrameHandler extends SimpleChannelInboundHandler<Tex
tWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) th
rows Exception {
            // Handle text frame
        }
    }

    public static final class BinaryFrameHandler extends SimpleChannelInboundHandler<B
inaryWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, BinaryWebSocketFrame msg)
throws Exception {
            // Handle binary frame
        }
    }

    public static final class ContinuationFrameHandler extends SimpleChannelInboundHan
dler<ContinuationWebSocketFrame> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, ContinuationWebSocketFrame
 msg) throws Exception {
            // Handle continuation frame
        }
    }
}
```

1. 添加 HttpObjectAggregator 用于提供在握手时聚合 HttpRequest
2. 添加 WebSocketServerProtocolHandler 用于处理色好给你寄握手如果请求是发送到"/websocket." 端点，当升级完成后，它将会处理Ping, Pong 和 Close 帧
3. TextFrameHandler 将会处理 TextWebSocketFrames
4. BinaryFrameHandler 将会处理 BinaryWebSocketFrames
5. ContinuationFrameHandler 将会处理ContinuationWebSocketFrames

加密 WebSocket 只需插入 SslHandler 到作为 pipline 第一个 ChannelHandler

详见 Chapter 11 WebSocket

SPDY

SPDY（读作“SPeeDY”）是Google 开发的基于 TCP 的应用层协议，用以最小化网络延迟，提升网络速度，优化用户的网络使用体验。SPDY 并不是一种用于替代 HTTP 的协议，而是对 HTTP 协议的增强。SPDY 实现技术：

- 压缩报头
- 加密所有
- 多路复用连接
- 提供支持不同的传输优先级

SPDY 主要有5个版本：

- 1 - 初始化版本，但没有使用
- 2 - 新特性，包含服务器推送
- 3 - 新特性包含流控制和更新压缩
- 3.1 - 会话层流程控制
- 4.0 - 流量控制，并与 HTTP 2.0 更加集成

SPDY 被很多浏览器支持，包括 Google Chrome, Firefox, 和 Opera

Netty 支持 版本 2 和 3 （包含3.1）的支持。这些版本被广泛应用，可以支持更多的用户。更多内容详见 Chapter 12

空闲连接以及超时

检测空闲连接和超时是为了及时释放资源。常见的方法发送消息用于测试一个不活跃的连接来,通常称为"心跳",到远端来确定它是否还活着。(一个更激进的方法是简单地断开那些指定的时间间隔的不活跃的连接)。

处理空闲连接是一项常见的任务,Netty 提供了几个 ChannelHandler 实现此目的。表8.4概述。

Table 8.4 ChannelHandlers for idle connections and timeouts

名称	描述
IdleStateHandler	如果连接闲置时间过长，则会触发 IdleStateEvent 事件。在 ChannelInboundHandler 中可以覆盖 userEventTriggered(...) 方法来处理 IdleStateEvent。
ReadTimeoutHandler	在指定的时间间隔内没有接收到入站数据则会抛出 ReadTimeoutException 并关闭 Channel。ReadTimeoutException 可以通过覆盖 ChannelHandler 的 exceptionCaught(…) 方法检测到。
WriteTimeoutHandler	WriteTimeoutException 可以通过覆盖 ChannelHandler 的 exceptionCaught(…) 方法检测到。

详细看下 IdleStateHandler，下面是一个例子，当超过60秒没有数据收到时，就会得到通知，此时就发送心跳到远端，如果没有回应，连接就关闭。

Listing 8.7 Sending heartbeats

```
public class IdleStateHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new IdleStateHandler(0, 0, 60, TimeUnit.SECONDS));  //1
        pipeline.addLast(new HeartbeatHandler());
    }

    public static final class HeartbeatHandler extends ChannelInboundHandlerAdapter {
        private static final ByteBuf HEARTBEAT_SEQUENCE = Unpooled.unreleasableBuffer(
                Unpooled.copiedBuffer("HEARTBEAT", CharsetUtil.ISO_8859_1));  //2

        @Override
        public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws E
xception {
            if (evt instanceof IdleStateEvent) {
                 ctx.writeAndFlush(HEARTBEAT_SEQUENCE.duplicate())
                         .addListener(ChannelFutureListener.CLOSE_ON_FAILURE);  //3
            } else {
                super.userEventTriggered(ctx, evt);  //4
            }
        }
    }
}
```

1. IdleStateHandler 将通过 IdleStateEvent 调用 userEventTriggered ，如果连接没有接收或发送数据超过60秒钟
2. 心跳发送到远端
3. 发送的心跳并添加一个侦听器，如果发送操作失败将关闭连接
4. 事件不是一个 IdleStateEvent 的话，就将它传递给下一个处理程序

总而言之,这个例子说明了如何使用 IdleStateHandler 测试远端是否还活着，如果不是就关闭连接释放资源。

解码分隔符和基于长度的协议

使用 Netty 时会遇到需要解码以分隔符和长度为基础的协议，本节讲解Netty 如何解码这些协议。

分隔符协议

经常需要处理分隔符协议或创建基于它们的协议，例如SMTP、POP3、IMAP、Telnet等等。Netty 附带的解码器可以很容易的提取一些序列分隔：

Table 8.5 Decoders for handling delimited and length-based protocols

名称	描述
DelimiterBasedFrameDecoder	接收ByteBuf由一个或多个分隔符拆分，如NUL或换行符
LineBasedFrameDecoder	接收ByteBuf以分割线结束，如"\n"和"\r\n"

下图显示了使用"\r\n"分隔符的处理：



1. 字节流
2. 第一帧
3. 第二帧

Figure 8.5 Handling delimited frames

下面展示了如何用 LineBasedFrameDecoder 处理

Listing 8.8 Handling line-delimited frames

```
public class LineBasedHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new LineBasedFrameDecoder(65 * 1024));   //1
        pipeline.addLast(new FrameHandler());  //2
    }

    public static final class FrameHandler extends SimpleChannelInboundHandler<ByteBuf
> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Except
ion {  //3
            // Do something with the frame
        }
    }
}
```

1. 添加一个 LineBasedFrameDecoder 用于提取帧并把数据包转发到下一个管道中的处理程序,在这种情况下就是 FrameHandler
2. 添加 FrameHandler 用于接收帧
3. 每次调用都需要传递一个单帧的内容

使用 DelimiterBasedFrameDecoder 可以方便处理特定分隔符作为数据结构体的这类情况。如下：

- 传入的数据流是一系列的帧，每个由换行（“\n”）分隔
- 每帧包括一系列项目，每个由单个空格字符分隔
- 一帧的内容代表一个“命令”：一个名字后跟一些变量参数

清单8.9中显示了的实现的方式。定义以下类：

- 类 Cmd - 存储帧的内容，其中一个 ByteBuf 用于存名字，另外一个存参数
- 类 CmdDecoder - 从重写方法 decode() 中检索一行，并从其内容中构建一个 Cmd 的实例
- 类 CmdHandler - 从 CmdDecoder 接收解码 Cmd 对象和对它的一些处理。

所以关键的解码器是扩展了 LineBasedFrameDecoder

Listing 8.9 Decoder for the command and the handler

```
public class CmdHandlerInitializer extends ChannelInitializer<Channel> {

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new CmdDecoder(65 * 1024));//1
```

```
        pipeline.addLast(new CmdHandler()); //2
    }

    public static final class Cmd { //3
        private final ByteBuf name;
        private final ByteBuf args;

        public Cmd(ByteBuf name, ByteBuf args) {
            this.name = name;
            this.args = args;
        }

        public ByteBuf name() {
            return name;
        }

        public ByteBuf args() {
            return args;
        }
    }

    public static final class CmdDecoder extends LineBasedFrameDecoder {
        public CmdDecoder(int maxLength) {
            super(maxLength);
        }

        @Override
        protected Object decode(ChannelHandlerContext ctx, ByteBuf buffer) throws Exce
ption {
            ByteBuf frame =  (ByteBuf) super.decode(ctx, buffer); //4
            if (frame == null) {
                return null; //5
            }
            int index = frame.indexOf(frame.readerIndex(), frame.writerIndex(), (byte)
 ' ');  //6
            return new Cmd(frame.slice(frame.readerIndex(), index), frame.slice(index
+1, frame.writerIndex())); //7
        }
    }

    public static final class CmdHandler extends SimpleChannelInboundHandler<Cmd> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, Cmd msg) throws Exception
{
            // Do something with the command  //8
        }
    }
}
```

1. 添加一个 CmdDecoder 到管道；将提取 Cmd 对象和转发到在管道中的下一个处理器
2. 添加 CmdHandler 将接收和处理 Cmd 对象

3. 命令也是 POJO
4. super.decode() 通过结束分隔从 ByteBuf 提取帧
5. frame 是空时，则返回 null
6. 找到第一个空字符的索引。首先是它的命令名；接下来是参数的顺序
7. 从帧先于索引以及它之后的片段中实例化一个新的 Cmd 对象
8. 处理通过管道的 Cmd 对象

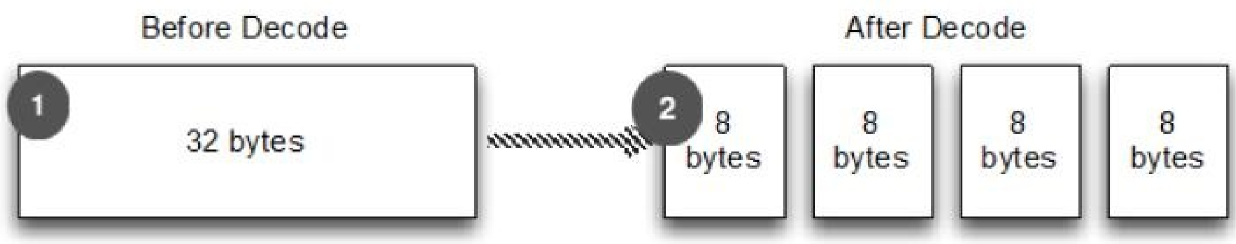
基于长度的协议

基于长度的协议协议在帧头文件里定义了一个帧编码的长度,而不是结束位置用一个特殊的分隔符来标记。表8.6列出了 Netty 提供的两个解码器，用于处理这种类型的协议。

Table 8.6 Decoders for length-based protocols

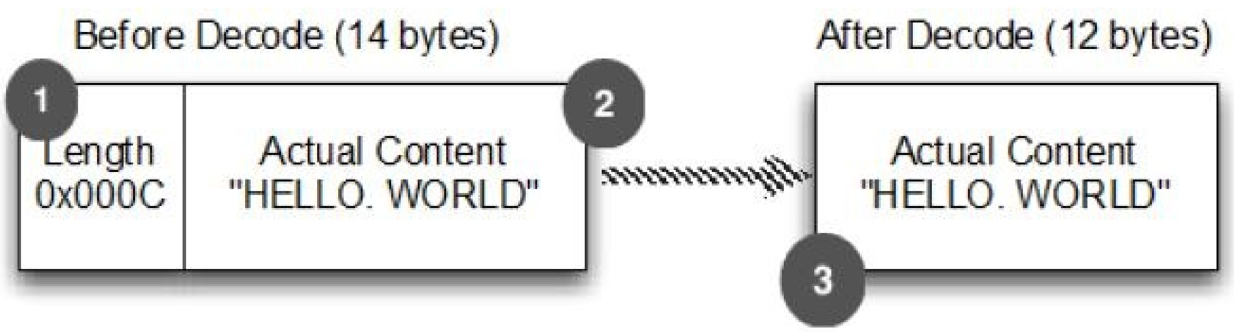
名称	描述
FixedLengthFrameDecoder	提取固定长度
LengthFieldBasedFrameDecoder	读取头部长度并提取帧的长度

如下图所示，FixedLengthFrameDecoder 的操作是提取固定长度每帧8字节



1. 字节流 stream
2. 4个帧，每个帧8个字节

大部分时候帧的大小被编码在头部，这种情况可以使用LengthFieldBasedFrameDecoder，它会读取头部长度并提取帧的长度。下图显示了它是如何工作的：



1. 长度 "0x000C" (12) 被编码在帧的前两个字节
2. 后面的12个字节就是内容

3. 提取没有头文件的帧内容

Figure 8.7 Message that has frame size encoded in the header

LengthFieldBasedFrameDecoder 提供了几个构造函数覆盖各种各样的头长字段配置情况。清单8.10显示了使用三个参数的构造函数是maxFrameLength,lengthFieldOffset lengthFieldLength。在这 情况下,帧的长度被编码在帧的前8个字节。

Listing 8.10 Decoder for the command and the handler

```
public class LengthBasedInitializer extends ChannelInitializer<Channel> {
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(
        new LengthFieldBasedFrameDecoder(65 * 1024, 0, 8)); //1
        pipeline.addLast(new FrameHandler()); //2
    }

    public static final class FrameHandler
        extends SimpleChannelInboundHandler<ByteBuf> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx,
        ByteBuf msg) throws Exception {
        // Do something with the frame //3
        }
    }
}
```

1. 添加一个 LengthFieldBasedFrameDecoder ,用于提取基于帧编码长度8个字节的帧。
2. 添加一个 FrameHandler 用来处理每帧
3. 处理帧数据

总而言之,本部分探讨了 Netty 提供的编解码器支持协议,包括定义特定的分隔符的字节流的结构或协议帧的长度。这些编解码器非常有用。

编写大型数据

由于网络的原因，如何有效的写大数据在异步框架是一个特殊的问题。因为写操作是非阻塞的，即便是在数据不能写出时,只是通知 ChannelFuture 完成了。当这种情况发生时,你必须停止写操作或面临内存耗尽的风险。所以写时,会产生大量的数据,我们需要做好准备来处理的这种情况下的缓慢的连接远端导致延迟释放内存的问题你。作为一个例子让我们考虑写一个文件的内容到网络。

在我们的讨论传输(见4.2节)时，我们提到了 NIO 的“zero-copy（零拷贝）”功能,消除移动一个文件的内容从文件系统到网络堆栈的复制步骤。所有这一切发生在 Netty 的核心,因此所有所需的应用程序代码是使用 interface FileRegion 的实现,在 Netty 的API 文档中定义如下为一个通过 Channel 支持 zero-copy 文件传输的文件区域。

下面演示了通过 zero-copy 将文件内容从 FileInputStream 创建 DefaultFileRegion 并写入 使用 Channel

Listing 8.11 Transferring file contents with FileRegion

```
FileInputStream in = new FileInputStream(file); //1
FileRegion region = new DefaultFileRegion(in.getChannel(), 0, file.length()); //2

channel.writeAndFlush(region).addListener(new ChannelFutureListener() { //3
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        if (!future.isSuccess()) {
            Throwable cause = future.cause(); //4
            // Do something
        }
    }
});
```

1. 获取 FileInputStream
2. 创建一个新的 DefaultFileRegion 用于文件的完整长度
3. 发送 DefaultFileRegion 并且注册一个 ChannelFutureListener
4. 处理发送失败

只是看到的例子只适用于直接传输一个文件的内容,没有执行的数据应用程序的处理。在相反的情况下,将数据从文件系统复制到用户内存是必需的,您可以使用 ChunkedWriteHandler。这个类提供了支持异步写大数据流不引起高内存消耗。

这个关键是 interface ChunkedInput，实现如下：

名称	描述
ChunkedFile	当你使用平台不支持 zero-copy 或者你需要转换数据，从文件中一块一块的获取数据
ChunkedNioFile	与 ChunkedFile 类似，处理使用了NIOFileChannel
ChunkedStream	从 InputStream 中一块一块的转移内容
ChunkedNioStream	从 ReadableByteChannel 中一块一块的转移内容

清单 8.12 演示了使用 ChunkedStream,实现在实践中最常用。 所示的类被实例化一个 File 和一个 SslContext。当 initChannel() 被调用来初始化显示的处理程序链的通道。

当通道激活时，WriteStreamHandler 从文件一块一块的写入数据作为ChunkedStream。最后将数据通过 SslHandler 加密后传播。

Listing 8.12 Transfer file content with FileRegion

```
public class ChunkedWriteHandlerInitializer extends ChannelInitializer<Channel> {
    private final File file;
    private final SslContext sslCtx;

    public ChunkedWriteHandlerInitializer(File file, SslContext sslCtx) {
        this.file = file;
        this.sslCtx = sslCtx;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new SslHandler(sslCtx.createEngine()); //1
        pipeline.addLast(new ChunkedWriteHandler());//2
        pipeline.addLast(new WriteStreamHandler());//3
    }

    public final class WriteStreamHandler extends ChannelInboundHandlerAdapter {  //4

        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            super.channelActive(ctx);
            ctx.writeAndFlush(new ChunkedStream(new FileInputStream(file)));
        }
    }
}
```

1. 添加 SslHandler 到 ChannelPipeline.
2. 添加 ChunkedWriteHandler 用来处理作为 ChunkedInput 传进的数据
3. 当连接建立时，WriteStreamHandler 开始写文件的内容
4. 当连接建立时，channelActive() 触发使用 ChunkedInput 来写文件的内容 (插图显示了

FileInputStream;也可以使用任何 InputStream)

ChunkedInput 所有被要求使用自己的 *ChunkedInput* 实现，是安装*ChunkedWriteHandler* 在管道中

在本节中,我们讨论

* 如何采用zero-copy（零拷贝）功能高效地传输文件
* 如何使用 ChunkedWriteHandler 编写大型数据而避免 OutOfMemoryErrors 错误。

在下一节中我们将研究几种不同方法来序列化 POJO。

序列化数据

JDK 提供了 ObjectOutputStream 和 ObjectInputStream 通过网络将原始数据类型和 POJO 进行序列化和反序列化。API并不复杂,可以应用到任何对象,支持 java.io.Serializable 接口。但它也不是非常高效的。在本节中,我们将看到 Netty 所提供的。

JDK 序列化

如果程序与端对端间的交互是使用 ObjectOutputStream 和 ObjectInputStream,并且主要面临的问题是兼容性,那么, JDK 序列化 是不错的选择。

表8.8列出了序列化类,Netty 提供了与 JDK 的互操作。

Table 8.8 JDK Serialization codecs

名称	描述
CompatibleObjectDecoder	该解码器使用 JDK 序列化,用于与非 Netty 进行互操作。
CompatibleObjectEncoder	该编码器使用 JDK 序列化,用于与非 Netty 进行互操作。
ObjectDecoder	基于 JDK 序列化来使用自定义序列化解码。外部依赖被排除在外时,提供了一个速度提升。否则选择其他序列化实现
ObjectEncoder	基于 JDK 序列化来使用自定义序列化编码。外部依赖被排除在外时,提供了一个速度提升。否则选择其他序列化实现

JBoss Marshalling 序列化

如果可以使用外部依赖 JBoss Marshalling 是个明智的选择。比 JDK 序列化快3倍且更加简练。

JBoss Marshalling 是另一个序列化 API,修复的许多 JDK序列化 API 中发现的问题,它与 java.io.Serializable 完全兼容。并添加了一些新的可调参数和附加功能,所有这些都可插入通过工厂配置外部化,类/实例查找表,类决议,对象替换,等等)

下表展示了 Netty 支持 JBoss Marshalling 的编解码器。

Table 8.9 JBoss Marshalling codecs

名称	描述
CompatibleMarshallingDecoder	为了与使用 JDK 序列化的端对端间兼容。
CompatibleMarshallingEncoder	为了与使用 JDK 序列化的端对端间兼容。
MarshallingDecoder	使用自定义序列化用于解码，必须使用

MarshallingEncoder MarshallingEncoder | 使用自定义序列化用于编码，必须使用 MarshallingDecoder

下面展示了使用 MarshallingDecoder 和 MarshallingEncoder

Listing 8.13 Using JBoss Marshalling

```
public class MarshallingInitializer extends ChannelInitializer<Channel> {

    private final MarshallerProvider marshallerProvider;
    private final UnmarshallerProvider unmarshallerProvider;

    public MarshallingInitializer(UnmarshallerProvider unmarshallerProvider,
                                  MarshallerProvider marshallerProvider) {
        this.marshallerProvider = marshallerProvider;
        this.unmarshallerProvider = unmarshallerProvider;
    }
    @Override
    protected void initChannel(Channel channel) throws Exception {
        ChannelPipeline pipeline = channel.pipeline();
        pipeline.addLast(new MarshallingDecoder(unmarshallerProvider));
        pipeline.addLast(new MarshallingEncoder(marshallerProvider));
        pipeline.addLast(new ObjectHandler());
    }

    public static final class ObjectHandler extends SimpleChannelInboundHandler<Serial
izable> {
        @Override
        public void channelRead0(ChannelHandlerContext channelHandlerContext, Serializ
able serializable) throws Exception {
            // Do something
        }
    }
}
```

ProtoBuf 序列化

ProtoBuf 来自谷歌，并且开源了。它使编解码数据更加紧凑和高效。它已经绑定各种编程语言,使它适合跨语言项目。

下表展示了 Netty 支持 ProtoBuf 的 ChannelHandler 实现。

Table 8.10 ProtoBuf codec

名称	描述
ProtobufDecoder	使用 ProtoBuf 来解码消息
ProtobufEncoder	使用 ProtoBuf 来编码消息
ProtobufVarint32FrameDecoder	在消息的整型长度域中，通过 "Base 128 Varints"将接收到的 ByteBuf 动态的分割

用法见下面

Listing 8.14 Using Google Protobuf

```
public class ProtoBufInitializer extends ChannelInitializer<Channel> {

    private final MessageLite lite;

    public ProtoBufInitializer(MessageLite lite) {
        this.lite = lite;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new ProtobufVarint32FrameDecoder());
        pipeline.addLast(new ProtobufEncoder());
        pipeline.addLast(new ProtobufDecoder(lite));
        pipeline.addLast(new ObjectHandler());
    }

    public static final class ObjectHandler extends SimpleChannelInboundHandler<Object
> {
        @Override
        public void channelRead0(ChannelHandlerContext ctx, Object msg) throws Excepti
on {
            // Do something with the object
        }
    }
}
```

1. 添加 ProtobufVarint32FrameDecoder 用来分割帧
2. 添加 ProtobufEncoder 用来处理消息的编码
3. 添加 ProtobufDecoder 用来处理消息的解码
4. 添加 ObjectHandler 用来处理解码了的消息

本章在这最后一节中,我们探讨了 Netty 支持的不同的序列化的专门的解码器和编码器。这些是标准 JDK 序列化 API,JBoss Marshalling 和谷歌ProtoBuf。

总结

Netty 的提供了编解码器和处理程序，可以组合和扩展来实现一个非常广泛的处理场景。此外,他们在许多大型系统被证明是健壮的组件。 请注意我们只介绍最常见的例子。API文档提供完整的描述。

引导

本章介绍：

- 引导客户端和服务器
- 从Channel引导客户端
- 添加 ChannelHandler
- 使用 ChannelOption 和属性

正如我们所见,ChannelPipeline 、ChannelHandler和编解码器提供工具,我们可以处理一个广泛的数据处理需求。但是你可能会问,“我创建了组件后,如何将其组装形成一个应用程序?”

答案是“bootstrapping（引导）”。到目前为止我们使用这个词有点模糊,时间可以来定义它。在最简单的条件下,引导就是配置应用程序的过程。但正如我们看到的,不仅仅如此；Netty 的引导客户端和服务器的类从网络基础设施使您的应用程序代码在后台可以连接和启动所有的组件。简而言之,引导使你的 Netty 应用程序完整。

Bootstrap 类型

Netty 的包括两种不同类型的引导。而不仅仅是当作的“服务器”和“客户”的引导，更有用的是考虑他们的目的是支持的应用程序功能。从这个意义上讲,“服务器”应用程序把一个“父”管道接受连接和创建“子”管道,而“客户端”很可能只需要一个单一的、非“父”对所有网络交互的管道（对于无连接的比如 UDP 协议也是一样）。

如图9.1所示,两个引导实现自一个名为 AbstractBootstrap 的超类。

Figure 9.1 Bootstrap hierarchy

前面的章节介绍的许多我们共同关注的话题,同样适用于客户端和服务器。这些都是由 AbstractBootstrap 处理,从而防止重复的功能和代码。专业引导类可以完全专注于它们独特的需要关心的地方。

克隆引导类

我们经常需要创建多个通道具有相似或相同的设置。支持这种模式而不需要为每个通道创建和配置一个新的引导实例, *AbstractBootstrap* 已经被标记为 *Cloneable*。调用 *clone()* 在一个已经配置引导将返回另一个引导实例并且是立即可用。

注意,因为这将创建只是 *EventLoopGroup* 浅拷贝,后者将会共享所有的克隆管道。这是可以接受的,因为往往是克隆的管道是 *short-lived(*短暂的，典型示例是管道创建用于 *HTTP* 请求*)*

下面内容将会关注 Bootstrap 和 ServerBootstrap

引导客户端和无连接协议

当需要引导客户端或一些无连接协议时，需要使用Bootstrap类。 在本节中,我们将回顾可用的各种方法引导客户端,引导线程,和可用的管道实现。

客户端引导方法

下表是 Bootstrap 的常用方法，其中很多是继承自 AbstractBootstrap。

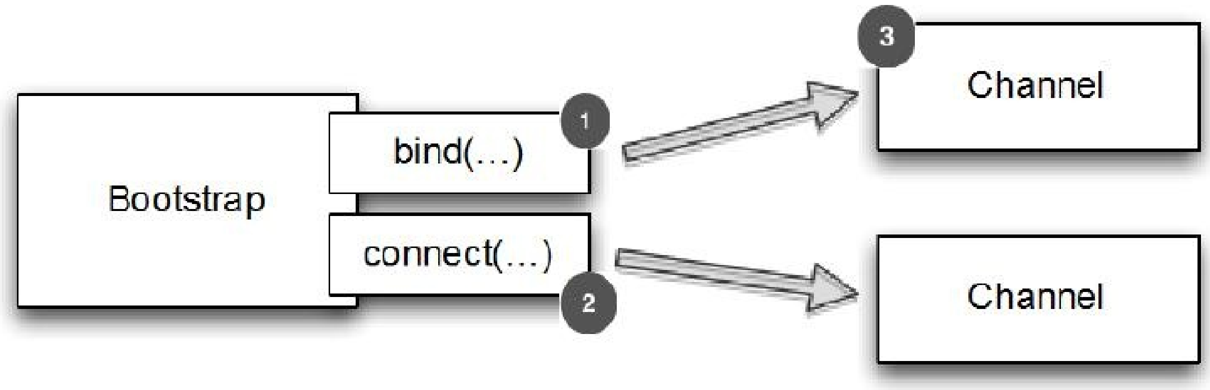
Table 9.1 Bootstrap methods

名称	描述
group	设置 EventLoopGroup 用于处理所有的 Channel 的事件
channel channelFactory	channel() 指定 Channel 的实现类。如果类没有提供一个默认的构造函数,你可以调用 channelFactory() 来指定一个工厂类被 bind() 调用。
localAddress	指定应该绑定到本地地址 Channel。如果不提供,将由操作系统创建一个随机的。或者,您可以使用 bind() 或 connect()指定localAddress
option	设置 ChannelOption 应用于 新创建 Channel 的 ChannelConfig。这些选项将被 bind 或 connect 设置在通道,这取决于哪个被首先调用。这个方法在创建管道后没有影响。所支持 ChannelOption 取决于使用的管道类型。请参考9.6节和 ChannelConfig 的 API 文档 的 Channel 类型使用。
attr	这些选项将被 bind 或 connect 设置在通道,这取决于哪个被首先调用。这个方法在创建管道后没有影响。请参考9.6节。
handler	设置添加到 ChannelPipeline 中的 ChannelHandler 接收事件通知。
clone	创建一个当前 Bootstrap的克隆拥有原来相同的设置。
remoteAddress	设置远程地址。此外,您可以通过 connect() 指定
connect	连接到远端，返回一个 ChannelFuture, 用于通知连接操作完成
bind	将通道绑定并返回一个 ChannelFuture,用于通知绑定操作完成后,必须调用 Channel.connect() 来建立连接。

如何引导客户端

Bootstrap 类负责创建管道给客户或应用程序，利用无连接协议和在调用 bind() 或 connect() 之后。

下图展示了如何工作



1. 当 bind() 调用时，Bootstrap 将创建一个新的管道, 当 connect() 调用在 Channel 来建立连接
2. Bootstrap 将创建一个新的管道, 当 connect() 调用时
3. 新的 Channel

Figure 9.2 Bootstrap process

下面演示了引导客户端，使用的是 NIO TCP 传输

Listing 9.1 Bootstrapping a client

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap(); //1
bootstrap.group(group) //2
    .channel(NioSocketChannel.class) //3
    .handler(new SimpleChannelInboundHandler<ByteBuf>() { //4
        @Override
        protected void channeRead0(
            ChannelHandlerContext channelHandlerContext,
            ByteBuf byteBuf) throws Exception {
                System.out.println("Received data");
                byteBuf.clear();
            }
        });
ChannelFuture future = bootstrap.connect(
    new InetSocketAddress("www.manning.com", 80)); //5
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Connection established");
            } else {
                System.err.println("Connection attempt failed");
                channelFuture.cause().printStackTrace();
            }
        }
    });
```

1. 创建一个新的 Bootstrap 来创建和连接到新的客户端管道
2. 指定 EventLoopGroup
3. 指定 Channel 实现来使用
4. 设置处理器给 Channel 的事件和数据
5. 连接到远端主机

注意 Bootstrap 提供了一个“流利”语法——示例中使用的方法(除了connect()) 由 Bootstrap 返回实例本身的引用链接他们。

兼容性

Channel 的实现和 EventLoop 的处理过程在 EventLoopGroup 中必须兼容，哪些 Channel 是和 EventLoopGroup 是兼容的可以查看 API 文档。经验显示，相兼容的实现一般在同一个包下面，例如使用NioEventLoop，NioEventLoopGroup 和 NioServerSocketChannel 在一起。请注意，这些都是前缀“Nio”，然后不会用这些代替另一个实现和另一个前缀，如“Oio”，也就是说 OioEventLoopGroup 和NioServerSocketChannel 是不相容的。

Channel 和 EventLoopGroup 的 EventLoop 必须相容，例如NioEventLoop、NioEventLoopGroup、NioServerSocketChannel是相容的，但是 OioEventLoopGroup 和 NioServerSocketChannel 是不相容的。从类名可以看出前缀是“Nio”的只能和“Nio”的一起使用。

EventLoop 和 *EventLoopGroup*

记住,*EventLoop* 分配给该 *Channel* 负责处理 *Channel* 的所有操作。当你执行一个方法,该方法返回一个 *ChannelFuture* ，它将在 分配给 *Channel* 的 *EventLoop* 执行。

EventLoopGroup 包含许多 *EventLoops* 和分配一个 *EventLoop* 通道时注册。我们将在*15*章更详细地讨论这个话题。

清单9.2所示的结果,试图使用一个 Channel 类型与一个 EventLoopGroup 兼容。

Listing 9.2 Bootstrap client with incompatible EventLoopGroup

```
EventLoopGroup group = new NioEventLoopGroup();
Bootstrap bootstrap = new Bootstrap(); //1
bootstrap.group(group) //2
    .channel(OioSocketChannel.class) //3
    .handler(new SimpleChannelInboundHandler<ByteBuf>() { //4
        @Override
        protected void channelRead0(
            ChannelHandlerContext channelHandlerContext,
                    ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
            }
        });
ChannelFuture future = bootstrap.connect(
    new InetSocketAddress("www.manning.com", 80)); //5
future.syncUninterruptibly();
```

1. 创建新的 Bootstrap 来创建新的客户端管道
2. 注册 EventLoopGroup 用于获取 EventLoop
3. 指定要使用的 Channel 类。通知我们使用 NIO 版本用于 EventLoopGroup ， OIO 用于 Channel
4. 设置处理器用于管道的 I/O 事件和数据
5. 尝试连接到远端。当 NioEventLoopGroup 和 OioSocketChannel 不兼容时，会抛出 IllegalStateException 异常

IllegalStateException 显示如下：

Listing 9.3 IllegalStateException thrown because of invalid configuration

```
Exception in thread "main" java.lang.IllegalStateException: incompatible event loop
type: io.netty.channel.nio.NioEventLoop
at
io.netty.channel.AbstractChannel$AbstractUnsafe.register(AbstractChannel.java:5
71)
...
```

出现 IllegalStateException 的其他情况是，在 bind() 或 connect() 调用前 调用需要设置参数的方法调用失败时，包括：

- group()
- channel() 或 channnelFactory()
- handler()

handler() 方法尤为重要,因为这些 ChannelPipeline 需要适当配置。 一旦提供了这些参数,应用程序将充分利用 Netty 的能力。

引导服务器

服务器的引导共用了客户端引导的一些逻辑。

引导服务器的方法

下表显示了 ServerBootstrap 的方法

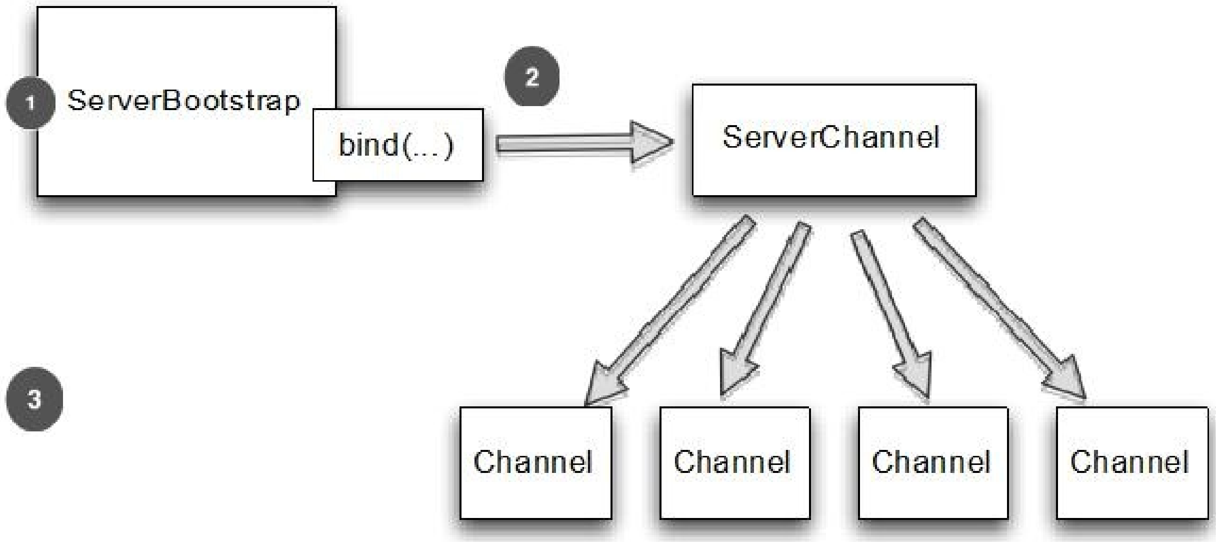
Table 9.2 Methods of ServerBootstrap‘

名称	描述
group	设置 EventLoopGroup 用于 ServerBootstrap。这个 EventLoopGroup 提供 ServerChannel 的 I/O 并且接收 Channel
channel channelFactory	channel() 指定 Channel 的实现类。如果管道没有提供一个默认的构造函数,你可以提供一个 ChannelFactory。
localAddress	指定 ServerChannel 实例化的类。如果不提供,将由操作系统创建一个随机的。或者,您可以使用 bind() 或 connect()指定localAddress
option	指定一个 ChannelOption 来用于新创建的 ServerChannel 的 ChannelConfig 。这些选项将被设置在管道的 bind() 或 connect(),这取决于谁首先被调用。在此调用这些方法之后设置或更改 ChannelOption 是无效的。所支持 ChannelOption 取决于使用的管道类型。请参考9.6节和 ChannelConfig 的 API 文档 的 Channel 类型使用。
childOption	当管道已被接受，指定一个 ChannelOption 应用于 Channel 的 ChannelConfig。
attr	指定 ServerChannel 的属性。这些属性可以被 管道的 bind() 设置。当调用 bind() 之后，修改它们不会生效。
childAttr	应用属性到接收到的管道上。后续调用没有效果。
handler	设置添加到 ServerChannel 的 ChannelPipeline 中的 ChannelHandler。 具体详见 childHandler() 描述
childHandler	设置添加到接收到的 Channel 的 ChannelPipeline 中的 ChannelHandler。handler() 和 childHandler()之间的区别是前者是接收和处理ServerChannel，同时 childHandler() 添加处理器用于处理和接收 Channel。后者代表一个套接字绑定到一个远端。
clone	克隆 ServerBootstrap 用于连接到不同的远端，通过设置相同的原始 ServerBoostrap。
bind	绑定 ServerChannel 并且返回一个 ChannelFuture,用于 通知连接操作完成了（结果可以是成功或者失败）

如何引导一个服务器

ServerBootstrap 中的 childHandler(), childAttr() 和 childOption() 是常用的服务器应用的操作。具体来说,ServerChannel实现负责创建子 Channel,它代表接受连接。因此 引导 ServerChannel 的 ServerBootstrap ,提供这些方法来简化接收的 Channel 对 ChannelConfig 应用设置的任务。

图9.3显示了 ServerChannel 创建 ServerBootstrap 在 bind(),后者管理大量的子 Channel。



1. 当调用 bind() 后 ServerBootstrap 将创建一个新的管道，这个管道将会在绑定成功后接收子管道
2. 接收新连接给每个子管道
3. 接收连接的 Channel

Figure 9.3 ServerBootstrap

记住 child* 的方法都是操作在子的 Channel，被 ServerChannel 管理。

清单9.4 ServerBootstrap 时会创建一个 NioServerSocketChannel实例 bind() 。这个 NioServerChannel 负责接受新连接和创建NioSocketChannel 实例。

Listing 9.4 Bootstrapping a server

```
NioEventLoopGroup group = new NioEventLoopGroup();
ServerBootstrap bootstrap = new ServerBootstrap(); //1
bootstrap.group(group) //2
    .channel(NioServerSocketChannel.class) //3
    .childHandler(new SimpleChannelInboundHandler<ByteBuf>() { //4
        @Override
        protected void channelRead0(ChannelHandlerContext ctx,
            ByteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
                byteBuf.clear();
            }
        }
    );
ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080)); //5
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture)
        throws Exception {
            if (channelFuture.isSuccess()) {
                System.out.println("Server bound");
            } else {
                System.err.println("Bound attempt failed");
                channelFuture.cause().printStackTrace();
            }
        }
    }
);
```

1. 创建要给新的 ServerBootstrap 来创建新的 SocketChannel 管道并绑定他们
2. 指定 EventLoopGroup 用于从注册的 ServerChannel 中获取EventLoop 和接收到的管道
3. 指定要使用的管道类
4. 设置子处理器用于处理接收的管道的 I/O 和数据
5. 通过配置引导来绑定管道

从 Channel 引导客户端

有时你可能需要引导客户端 Channel 从另一个 Channel。这可能发生,如果您正在编写一个代理或从其他系统需要检索数据。后一种情况是常见的,因为许多 Netty 的应用程序集成现有系统,例如 web 服务或数据库。

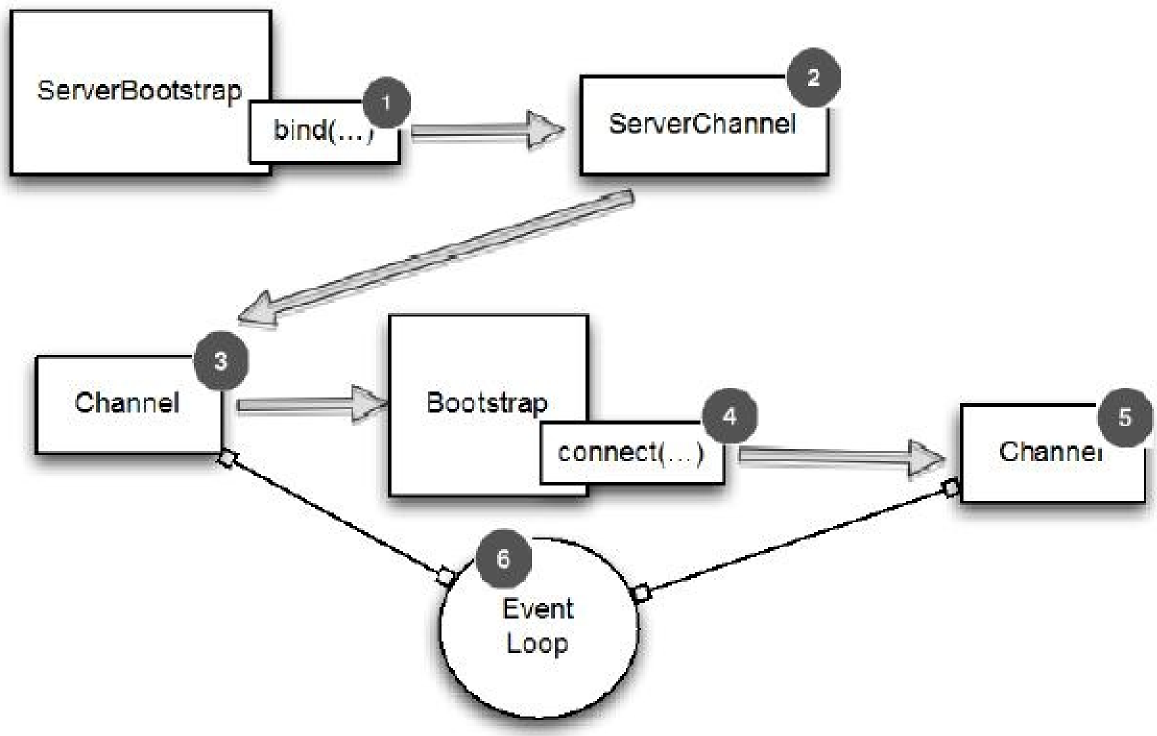
你当然可以创建一个新的 Bootstrap 并使用它如9.2.1节所述,这个解决方案不一定有效。至少,你需要创建另一个 EventLoop 给新客户端 Channel 的,并且 Channel 将会需要在不同的 Thread 间进行上下文切换。

幸运的是,由于 EventLoop 继承自 EventLoopGroup ,您可以通过传递 接收到的 Channel 的 EventLoop 到 Bootstrap 的 group() 方法。这允许客户端 Channel 来操作 相同的 EventLoop,这样就能消除了额外的线程创建和所有相关的上下文切换的开销。

为什么共享 *EventLoop* 呢?

当你分享一个 *EventLoop* ,你保证所有 *Channel* 分配给 *EventLoop* 将使用相同的线程,消除上下文切换和相关的开销。*(请记住,一个EventLoop分配给一个线程执行操作。)*

共享一个 EventLoop 描述如下：



1. 当 bind() 调用时，ServerBootstrap 创建一个新的ServerChannel 。 当绑定成功后，这个管道就能接收子管道了
2. ServerChannel 接收新连接并且创建子管道来服务它们

3. Channel 用于接收到的连接
4. 管道自己创建了 Bootstrap，用于当 connect() 调用时创建一个新的管道
5. 新管道连接到远端
6. 在 EventLoop 接收通过 connect() 创建后就在管道间共享

Figure 9.4 EventLoop shared between channels with ServerBootstrap and Bootstrap

实现 EventLoop 共享，包括设置 EventLoop 引导通过Bootstrap.eventLoop() 方法。这是清单9.5所示。

```
ServerBootstrap bootstrap = new ServerBootstrap(); //1
bootstrap.group(new NioEventLoopGroup(), //2
    new NioEventLoopGroup()).channel(NioServerSocketChannel.class) //3
        .childHandler(        //4
            new SimpleChannelInboundHandler<ByteBuf>() {
            ChannelFuture connectFuture;

            @Override
            public void channelActive(ChannelHandlerContext ctx) throws Exception {
                Bootstrap bootstrap = new Bootstrap();//5
                bootstrap.channel(NioSocketChannel.class) //6
                        .handler(new SimpleChannelInboundHandler<ByteBuf>() {  //7
                            @Override
                            protected void channelRead0(ChannelHandlerContext ctx, Byt
eBuf in) throws Exception {
                                System.out.println("Reveived data");
                            }
                        });
                bootstrap.group(ctx.channel().eventLoop()); //8
                connectFuture = bootstrap.connect(new InetSocketAddress("www.manning.c
om", 80));  //9
            }

            @Override
            protected void channelRead0(ChannelHandlerContext channelHandlerContext, B
yteBuf byteBuf) throws Exception {
                if (connectFuture.isDone()) {
                    // do something with the data  //10
                }
            }
        });
ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));  //11
future.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture channelFuture) throws Exception {
        if (channelFuture.isSuccess()) {
            System.out.println("Server bound");
        } else {
            System.err.println("Bound attempt failed");
            channelFuture.cause().printStackTrace();
        }
    }
});
```

1. 创建一个新的 ServerBootstrap 来创建新的 SocketChannel 管道并且绑定他们
2. 指定 EventLoopGroups 从 ServerChannel 和接收到的管道来注册并获取 EventLoops
3. 指定 Channel 类来使用
4. 设置处理器用于处理接收到的管道的 I/O 和数据
5. 创建一个新的 Bootstrap 来连接到远程主机
6. 设置管道类

7. 设置处理器来处理 I/O
8. 使用相同的 EventLoop 作为分配到接收的管道
9. 连接到远端
10. 连接完成处理业务逻辑 (比如, proxy)
11. 通过配置了的 Bootstrap 来绑定到管道

注意，新的 EventLoop 会创建一个新的 Thread。出于该原因，EventLoop 实例应该尽量重用。或者限制实例的数量来避免耗尽系统资源。

在一个引导中添加多个 ChannelHandler

在所有的例子代码中，我们在引导过程中通过 handler() 或childHandler() 都只添加了一个 ChannelHandler 实例，对于简单的程序可能足够，但是对于复杂的程序则无法满足需求。例如，某个程序必须支持多个协议，如 HTTP、WebSocket。若在一个 ChannelHandle r中处理这些协议将导致一个庞大而复杂的 ChannelHandler。Netty 通过添加多个 ChannelHandler，从而使每个 ChannelHandler 分工明确，结构清晰。

Netty 的一个优势是可以在 ChannelPipeline 中堆叠很多ChannelHandler 并且可以最大程度的重用代码。如何添加多个ChannelHandler 呢？Netty 提供 ChannelInitializer 抽象类用来初始化 ChannelPipeline 中的 ChannelHandler。ChannelInitializer是一个特殊的 ChannelHandler，通道被注册到 EventLoop 后就会调用ChannelInitializer，并允许将 ChannelHandler 添加到CHannelPipeline；完成初始化通道后，这个特殊的 ChannelHandler 初始化器会从 ChannelPipeline 中自动删除。

听起来很复杂，其实很简单，看下面代码：

Listing 9.6 Bootstrap and using ChannelInitializer

```
ServerBootstrap bootstrap = new ServerBootstrap();//1
bootstrap.group(new NioEventLoopGroup(), new NioEventLoopGroup())  //2
    .channel(NioServerSocketChannel.class)  //3
    .childHandler(new ChannelInitializerImpl()); //4
ChannelFuture future = bootstrap.bind(new InetSocketAddress(8080));  //5
future.sync();


final class ChannelInitializerImpl extends ChannelInitializer<Channel> {  //6
    @Override
    protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline(); //7
        pipeline.addLast(new HttpClientCodec());
        pipeline.addLast(new HttpObjectAggregator(Integer.MAX_VALUE));

    }
}
```

1. 创建一个新的 ServerBootstrap 来创建和绑定新的 Channel
2. 指定 EventLoopGroups 从 ServerChannel 和接收到的管道来注册并获取 EventLoops
3. 指定 Channel 类来使用
4. 设置处理器用于处理接收到的管道的 I/O 和数据
5. 通过配置的引导来绑定管道
6. ChannelInitializer 负责设置 ChannelPipeline

7. 实现 initChannel() 来添加需要的处理器到 ChannelPipeline。一旦完成了这方法 ChannelInitializer 将会从 ChannelPipeline 删除自身。

通过 ChannelInitializer, Netty 允许你添加你程序所需的多个 ChannelHandler 到 ChannelPipeline

使用Netty 的 ChannelOption 和属性

比较麻烦的是创建通道后不得不手动配置每个通道，为了避免这种情况，Netty 提供了 ChannelOption 来帮助引导配置。这些选项会自动应用到引导创建的所有通道，可用的各种选项可以配置底层连接的详细信息，如通道“keep-alive(保持活跃)”或“timeout(超时)”的特性。

Netty 应用程序通常会与组织或公司其他的软件进行集成，在某些情况下，Netty 的组件如 Channel 在 Netty 正常生命周期外使用； Netty 的提供了抽象 AttributeMap 集合,这是由 Netty 的管道和引导类,和 AttributeKey，常见类用于插入和检索属性值。属性允许您安全的关联任何数据项与客户端和服务器的 Channel。

例如,考虑一个服务器应用程序跟踪用户和 Channel 之间的关系。这可以通过存储用户 ID 作为 Channel 的一个属性。类似的技术可以用来路由消息到基于用户 ID 或关闭基于用户活动的一个管道。

清单9.7展示了如何使用 ChannelOption 配置 Channel 和一个属性来存储一个整数值。

Listing 9.7 Using Attributes

```
final AttributeKey<Integer> id = new AttributeKey<Integer>("ID");  //1

Bootstrap bootstrap = new Bootstrap(); //2
bootstrap.group(new NioEventLoopGroup()) //3
        .channel(NioSocketChannel.class) //4
        .handler(new SimpleChannelInboundHandler<ByteBuf>() { //5
            @Override
            public void channelRegistered(ChannelHandlerContext ctx) throws Exception
{
               Integer idValue = ctx.channel().attr(id).get();  //6
                // do something  with the idValue
            }

            @Override
            protected void channelRead0(ChannelHandlerContext channelHandlerContext, B
yteBuf byteBuf) throws Exception {
                System.out.println("Reveived data");
            }
        });
bootstrap.option(ChannelOption.SO_KEEPALIVE, true).option(ChannelOption.CONNECT_TIMEOU
T_MILLIS, 5000);   //7
bootstrap.attr(id, 123456); //8

ChannelFuture future = bootstrap.connect(new InetSocketAddress("www.manning.com", 80))
;   //9
future.syncUninterruptibly();
```

1. 新建一个 AttributeKey 用来存储属性值
2. 新建 Bootstrap 用来创建客户端管道并连接他们
3. 指定 EventLoopGroups 从和接收到的管道来注册并获取 EventLoop
4. 指定 Channel 类
5. 设置处理器来处理管道的 I/O 和数据
6. 检索 AttributeKey 的属性及其值
7. 设置 ChannelOption 将会设置在管道在连接或者绑定
8. 存储 id 属性
9. 通过配置的 Bootstrap 来连接到远程主机

关闭之前已经引导的客户端或服务器

引导您的应用程序启动并运行,但是迟早你也需要关闭它。当然你可以让 JVM处理所有退出但这不会满足“优雅”的定义，是指干净地释放资源。关闭一个Netty 的应用程序并不复杂,但有几件事要记住。

主要是记住关闭 EventLoopGroup,将处理任何悬而未决的事件和任务并随后释放所有活动线程。这只是一种叫EventLoopGroup.shutdownGracefully()。这个调用将返回一个 Future 用来通知关闭完成。注意,shutdownGracefully()也是一个异步操作,所以你需要阻塞,直到它完成或注册一个侦听器直到返回的 Future 来通知完成。

清单9.9定义了“优雅地关闭”

Listing 9.9 Graceful shutdown

```
EventLoopGroup group = new NioEventLoopGroup() //1
Bootstrap bootstrap = new Bootstrap(); //2
bootstrap.group(group)
    .channel(NioSocketChannel.class);
...
...
Future<?> future = group.shutdownGracefully(); //3
// block until the group has shutdown
future.sync();
```

1. 创建 EventLoopGroup 用于处理 I/O
2. 创建一个新的 Bootstrap 并且配置他
3. 最终优雅的关闭 EventLoopGroup 释放资源。这个也会关闭中当前使用的 Channel

或者,您可以调用 Channel.close() 显式地在所有活动管道之前调用 EventLoopGroup.shutdownGracefully()。但是在所有情况下,记得关闭EventLoopGroup 本身

总结

在本章中,您了解了如何引导基于 Netty 服务器和客户端应用程序(包括那些使用无连接协议),如何指定管道的配置选项,以及如何使用属性信息附加到一个管道。

在下一章,我们将研究如何测试你 ChannelHandler 实现以确保其正确性。

单元测试

本章介绍

- 单元测试
- EmbeddedChannel

学会了使用一个或多个 ChannelHandler 处理接收/发送数据消息，但是如何测试它们呢？Netty 提供了2个额外的类使得测试 ChannelHandler变得很容易，本章讲解如何测试 Netty 程序。测试使用 JUnit4，如果不会用可以慢慢了解。JUnit4 很简单，但是功能很强大。

本章将重点讲解测试已实现的 ChannelHandler 和编解码器

总览

我们已经知道,ChannelHandler 实现可以串联在一起,以构建ChannelPipeline 的处理逻辑。我们先前解释说,这个设计方法 支持潜在的复杂的分解处理成小和可重用的组件,其中每个一个定义良好的处理任务或步骤。在这一章里,我们将展示它简化了测试。

Netty 的促进 ChannelHandler 的测试通过的所谓“嵌入式”传输。这是由一个特殊 Channel 实现,EmbeddedChannel,它提供了一个简单的方法通过管道传递事件。

想法很简单:你入站或出站数据写入一个E mbeddedChannel 然后检查是否达到 ChannelPipeline 的结束。这样你可以确定消息编码或解码和ChannelHandler 是否操作被触发。

在表10.1中列出了相关方法。

名称	职责
writeInbound	写一个入站消息到 EmbeddedChannel。 如果数据能从 EmbeddedChannel 通过 readInbound() 读到，则返回 true
readInbound	从 EmbeddedChannel 读到入站消息。任何返回遍历整个 ChannelPipeline。如果读取还没有准备，则此方法返回 null
writeOutbound	写一个出站消息到 EmbeddedChannel。 如果数据能从 EmbeddedChannel 通过 readOutbound() 读到，则返回 true
readOutbound	从 EmbeddedChannel 读到出站消息。任何返回遍历整个 ChannelPipeline。如果读取还没有准备，则此方法返回 null
Finish	如果从入站或者出站中能读到数据，标记 EmbeddedChannel 完成并且返回。这同时会调用 EmbeddedChannel 的关闭方法

测试入站和出站数据

处理入站数据由 *ChannelInboundHandler* 处理并且表示数据从远端读取。出站数据由 *ChannelOutboundHandler* 处理并且表示数据写入远端。 根据 *ChannelHandler* 测试你会选择 *writeInbound(),writeOutbound(),* 或者两者都有。

图10.1显示了数据流如何通过 ChannelPipeline 使用 EmbeddedChannel 的方法。

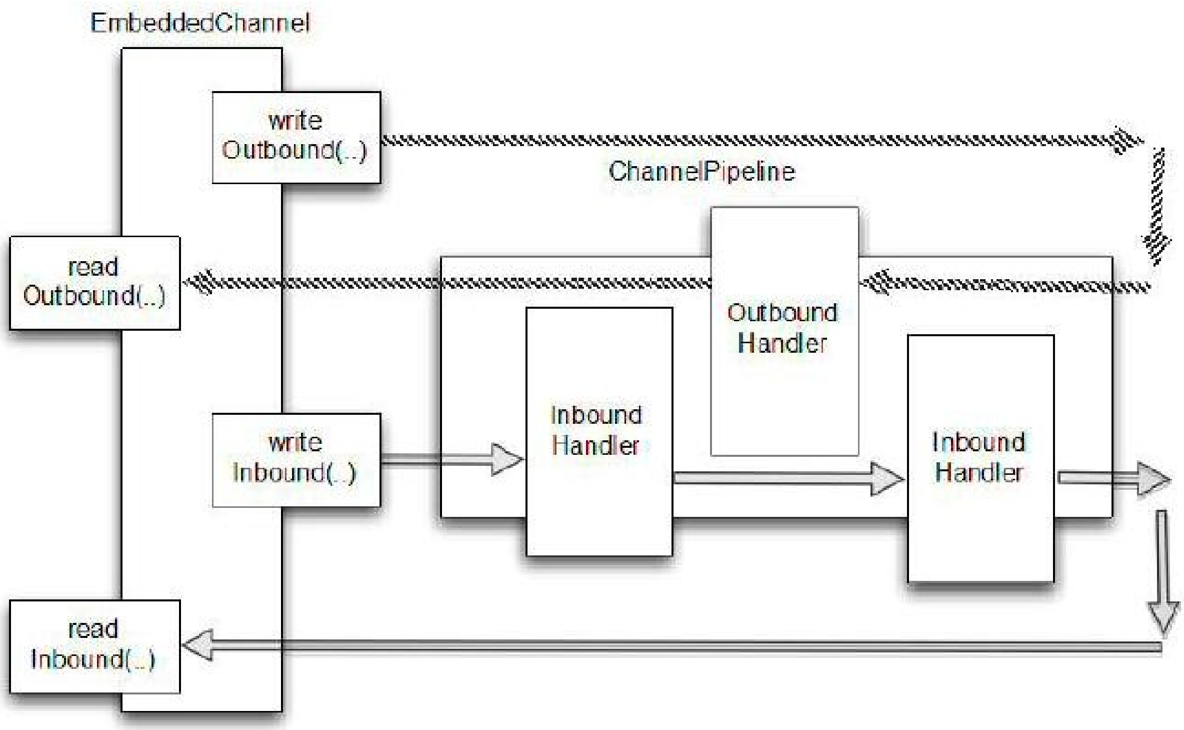


Figure 10.1 EmbeddedChannel data flow

如上图所示，使用 writeOutbound() 写消息到 Channel，消息在出站方法通过 ChannelPipeline，之后就可以使用 readOutbound() 读取消息。着同样使用与入站，使用 writeInbound() 和 readInbound()。处在

每种情况下,消息是通过 ChannelPipeline 并被有关ChannelInboundHandler 或 ChannelOutboundHandler 进行处理。如果消息是不消耗您可以使用 readInbound() 或 readOutbound() 适当的读到 Channel 处理后的消息。

让我们仔细看一下这两个场景,看看他们如何适用于测试您的应用程序逻辑。

测试 ChannelHandler

本节，将使用 EmbeddedChannel 来测试 ChannelHandler

测试入站消息

我们来编写一个简单的 ByteToMessageDecoder 实现，有足够的数据可以读取时将产生固定大小的包，如果没有足够的数据可以读取，则会等待下一个数据块并再次检查是否可以产生一个完整包。

如图所示，它可能会占用一个以上的“event”以获取足够的字节产生一个数据包，并将它传递到 ChannelPipeline 中的下一个 ChannelHandler，



Figure 10.2 Decoding via FixedLengthFrameDecoder

实现如下：

Listing 10.1 FixedLengthFrameDecoder implementation

```
public class FixedLengthFrameDecoder extends ByteToMessageDecoder { //1

    private final int frameLength;

    public FixedLengthFrameDecoder(int frameLength) { //2
        if (frameLength <= 0) {
            throw new IllegalArgumentException(
                    "frameLength must be a positive integer: " + frameLength);
        }
        this.frameLength = frameLength;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) thr
ows Exception {
        if (in.readableBytes() >= frameLength) { //3
            ByteBuf buf = in.readBytes(frameLength);//4
            out.add(buf); //5
        }
    }
}
```

1. 继承 ByteToMessageDecoder 用来处理入站的字节并将他们解码为消息
2. 指定产出的帧的长度
3. 检查是否有足够的字节用于读到下个帧
4. 从 ByteBuf 读取新帧
5. 添加帧到解码好的消息 List

下面是单元测试的例子，使用 EmbeddedChannel

Listing 10.2 Test the FixedLengthFrameDecoder

```
public class FixedLengthFrameDecoderTest {

    @Test    //1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer(); //2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(new FixedLengthFrameDecoder(3));
 //3
        Assert.assertFalse(channel.writeInbound(input.readBytes(2))); //4
        Assert.assertTrue(channel.writeInbound(input.readBytes(7)));

        Assert.assertTrue(channel.finish());  //5
        ByteBuf read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        Assert.assertNull(channel.readInbound());
        buf.release();
    }


    @Test
    public void testFramesDecoded2() {
        ByteBuf buf = Unpooled.buffer();
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();
```

```
        EmbeddedChannel channel = new EmbeddedChannel(new FixedLengthFrameDecoder(3));
        Assert.assertFalse(channel.writeInbound(input.readBytes(2)));
        Assert.assertTrue(channel.writeInbound(input.readBytes(7)));

        Assert.assertTrue(channel.finish());
        ByteBuf read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(3), read);
        read.release();

        Assert.assertNull(channel.readInbound());
        buf.release();
    }
}
```

1. 测试增加 @Test 注解
2. 新建 ByteBuf 并用字节填充它
3. 新增 EmbeddedChannel 并添加 FixedLengthFrameDecoder 用于测试
4. 写数据到 EmbeddedChannel
5. 标记 channel 已经完成
6. 读产生的消息并且校验

如上面代码，testFramesDecoded() 方法想测试一个 ByteBuf，这个ByteBuf 包含9个可读字节，被解码成包含了3个可读字节的 ByteBuf。你可能注意到，它写入9字节到通道是通过调用 writeInbound() 方法，之后再执行 finish() 来将 EmbeddedChannel 标记为已完成，最后调用 readInbound() 方法来获取 EmbeddedChannel 中的数据，直到没有可读字节。testFramesDecoded2() 方法采取同样的方式，但有一个区别就是入站ByteBuf分两步写的，当调用 writeInbound(input.readBytes(2)) 后返回 false 时，FixedLengthFrameDecoder 值会产生输出，至少有3个字节是可读，testFramesDecoded2() 测试的工作相当于 testFramesDecoded()。

Testing outbound messages

测试的处理出站消息类似于我们刚才看到的一切。这个例子将使用的实现 MessageToMessageEncoder:AbsIntegerEncoder。

- 当收到 flush() 它将从 ByteBuf 读取4字节整数并给每个执行Math.abs()。
- 每个整数接着写入 ChannelHandlerPipeline

图10.3显示了逻辑。

Figure 10.3 Encoding via AbsIntegerEncoder

示例如下：

Listing 10.3 AbsIntegerEncoder

```
public class AbsIntegerEncoder extends MessageToMessageEncoder<ByteBuf> {  //1
    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext, ByteBuf in, List<Object> out) throws Exception {
        while (in.readableBytes() >= 4) { //2
            int value = Math.abs(in.readInt());//3
            out.add(value);  //4
        }
    }
}
```

1. 继承 MessageToMessageEncoder 用于编码消息到另外一种格式
2. 检查是否有足够的字节用于编码
3. 读取下一个输入 ByteBuf 产出的 int 值，并计算绝对值
4. 写 int 到编码的消息 List

在前面的示例中,我们将使用 EmbeddedChannel 测试代码。清单10.4

Listing 10.4 Test the AbsIntegerEncoder

```
public class AbsIntegerEncoderTest {

    @Test   //1
    public void testEncoded() {
        ByteBuf buf = Unpooled.buffer();  //2
        for (int i = 1; i < 10; i++) {
            buf.writeInt(i * -1);
        }

        EmbeddedChannel channel = new EmbeddedChannel(new AbsIntegerEncoder());  //3
        Assert.assertTrue(channel.writeOutbound(buf)); //4

        Assert.assertTrue(channel.finish()); //5
        for (int i = 1; i < 10; i++) {
            Assert.assertEquals(i, channel.readOutbound());  //6
        }
        Assert.assertNull(channel.readOutbound());
    }
}
```

1. 用 @Test 标记
2. 新建 ByteBuf 并写入负整数
3. 新建 EmbeddedChannel 并安装 AbsIntegerEncoder 来测试
4. 写 ByteBuf 并预测 readOutbound() 产生的数据
5. 标记 channel 已经完成
6. 读取产生到的消息，检查负值已经编码为绝对值

测试异常处理

有时候传输的入站或出站数据不够，通常这种情况也需要处理，例如抛出一个异常。这可能是你错误的输入或处理大的资源或其他的异常导致。我们来写一个实现，如果输入字节超出限制长度就抛出TooLongFrameException，这样的功能一般用来防止资源耗尽。看下图：

在图10.4最大帧大小被设置为3个字节。



Figure 10.4 Decoding via FrameChunkDecoder

上图显示帧的大小被限制为3字节，若输入的字节超过3字节，则超过的字节被丢弃并抛出TooLongFrameException。在 ChannelPipeline 中的其他ChannelHandler 实现可以处理TooLongFrameException 或者忽略异常。处理异常在 ChannelHandler.exceptionCaught() 方法中完成，ChannelHandler 提供了一些具体的实现，看下面代码：

```
public class FrameChunkDecoder extends ByteToMessageDecoder {  //1

    private final int maxFrameSize;

    public FrameChunkDecoder(int maxFrameSize) {
        this.maxFrameSize = maxFrameSize;
    }

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out) thr
ows Exception {
        int readableBytes = in.readableBytes();  //2
        if (readableBytes > maxFrameSize)  {
            // discard the bytes   //3
            in.clear();
            throw new TooLongFrameException();
        }
        ByteBuf buf = in.readBytes(readableBytes); //4
        out.add(buf);  //5
    }
}
```

1. 继承 ByteToMessageDecoder 用于解码入站字节到消息
2. 指定最大需要的帧产生的体积
3. 如果帧太大就丢弃并抛出一个 TooLongFrameException 异常

4. 同时从 ByteBuf 读到新帧
5. 添加帧到解码消息 List

示例如下：

Listing 10.6 Testing FixedLengthFrameDecoder

```
public class FrameChunkDecoderTest {

    @Test    //1
    public void testFramesDecoded() {
        ByteBuf buf = Unpooled.buffer();  //2
        for (int i = 0; i < 9; i++) {
            buf.writeByte(i);
        }
        ByteBuf input = buf.duplicate();

        EmbeddedChannel channel = new EmbeddedChannel(new FrameChunkDecoder(3));  //3
        Assert.assertTrue(channel.writeInbound(input.readBytes(2)));  //4
        try {
            channel.writeInbound(input.readBytes(4)); //5
            Assert.fail();  //6
        } catch (TooLongFrameException e) {
            // expected
        }
        Assert.assertTrue(channel.writeInbound(input.readBytes(3)));  //7


        Assert.assertTrue(channel.finish());  //8

        ByteBuf read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.readSlice(2), read); //9
        read.release();

        read = (ByteBuf) channel.readInbound();
        Assert.assertEquals(buf.skipBytes(4).readSlice(3), read);
        read.release();

        buf.release();
    }
}
```

1. 使用 @Test 注解
2. 新建 ByteBuf 写入 9 个字节
3. 新建 EmbeddedChannel 并安装一个 FixedLengthFrameDecoder 用于测试
4. 写入 2 个字节并预测生产的新帧(消息)
5. 写一帧大于帧的最大容量 (3) 并检查一个 TooLongFrameException 异常
6. 如果异常没有被捕获，测试将失败。注意如果类实现 exceptionCaught() 并且处理了异常 exception，那么这里就不会捕捉异常

7. 写剩余的 2 个字节预测一个帧
8. 标记 channel 完成
9. 读到的产生的消息并且验证值。注意 assertEquals(Object,Object)测试使用 equals() 是否相当，不是对象的引用是否相当

即使我们使用 EmbeddedChannel 和 ByteToMessageDecoder。

应该指出的是,同样的可以做每个 ChannelHandler 的实现,将抛出一个异常。

乍一看,这看起来很类似于测试我们写在清单10.2中,但它有一个有趣的转折,即 TooLongFrameException 的处理。这里使用的 try/catch 块是 EmbeddedChannel 的一种特殊的特性。如果其中一个“write*"编写方法产生一个受控异常将被包装在一个 RuntimeException。这使得测试更加容易,如果异常处理的一部分处理。

总结

使用测试工具，如JUnit单元测试是一个非常有效的方式保证代码的正确性,提高其可维护性。在本章中,您了解了如何测试定制 ChannelHandler 来验证他们的工作。

在接下来的章节我们将专注于写 Netty “真实世界” 的应用程序。即使我们任何进一步的测试代码的例子，但希望你能记住我们的测试方法的探讨及其重要性。

WebSocket

本章涵盖了如下内容：

- WebSockets
- ChannelHandler, Decoder 和 Encoder
- 引导你的应用程序

real-time web（实时web）是一组技术和实践，使用户能够实时地接收 到作者发布的信息，而不需要用户用他们的软件定期检查更新源。

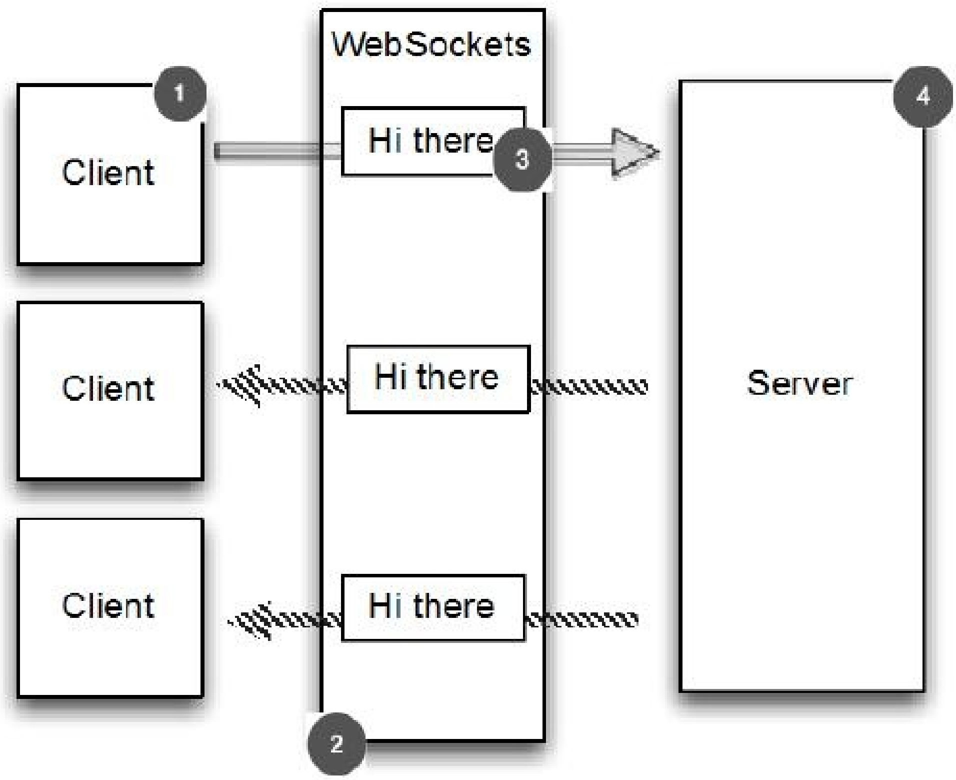
HTTP 的请求/响应的设计并不能满足实时的需求，而 WebSocket 协议从设计以来就提供双向数据传输，允许客户和服务器在任何时间发送消息，并要求它们能够异步处理消息。最新的浏览器都将 WebSockets 作为HTML5的一种客户端API来支持的。

Netty 中对于 WebSocket 的支持包括正在使用的所有主要的实现，所以在你的下一个应用程序中采用它会非常简单。像往常使用Netty一样，你可以充分利用这种协议，而不必担心其内部实现细节。 我们将通过开发基于 WebSocket 的实时聊天应用证明这一点。

WebSocket 程序示例

为了说明实时功能的特点，我们使用 WebSocket 协议来实现一个基于浏览器的实时聊天程序，就像你在 Facebook 中用文字聊天一样。但是我们这里要更进一步，我们要让不同的用户可以同时互相交谈。

程序逻辑如图 11.1 所示



#1客户端/用户连接到服务器，并且是聊天的一部分

#2聊天消息通过 WebSocket 进行交换

#3消息双向发送

#4服务器处理所有的客户端/用户

逻辑很简单：

- 1.客户端发送一个消息。
- 2.消息被广播到所有其他连接的客户端。

这正如你所想的聊天室的工作方式：每个人都可以跟其他人聊天。此例子将仅提供服务器端，浏览器充当客户端，通过访问网页来聊天。正如您接下来要看到的，WebSocket 让这一切变得简单。

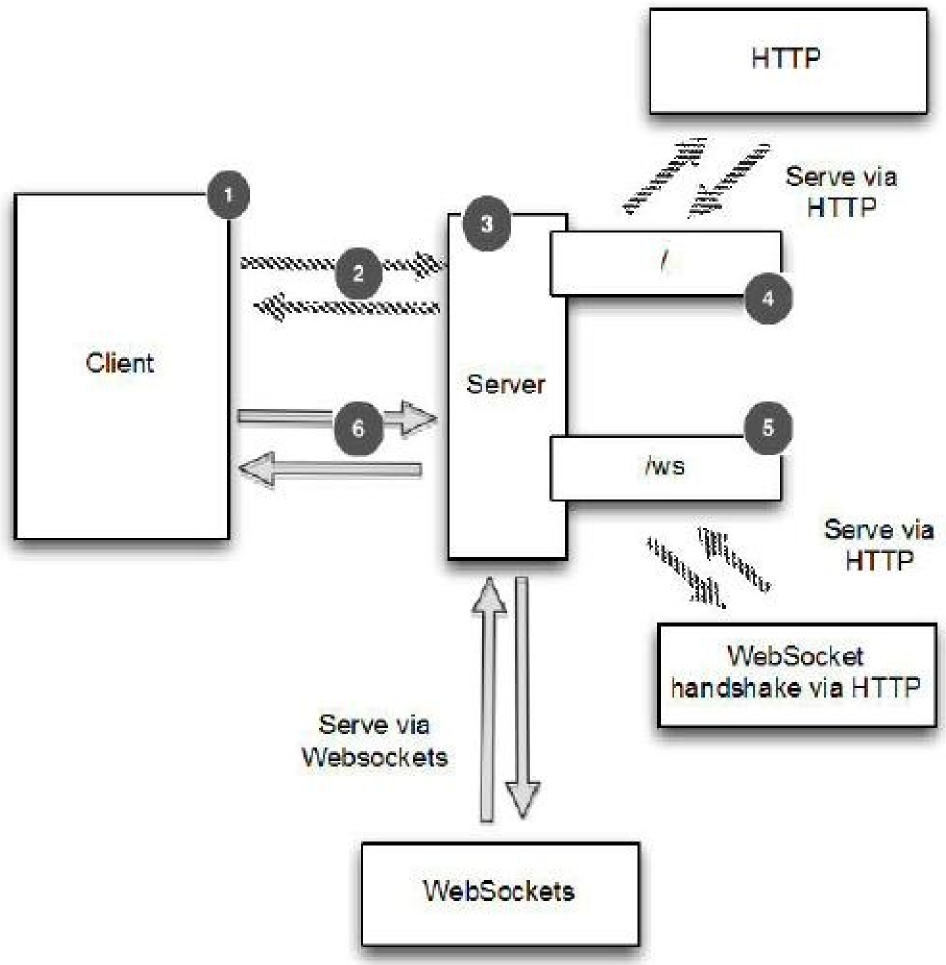
添加 WebSocket 支持

WebSocket 使用一种被称作“Upgrade handshake（升级握手）”的机制将标准的 HTTP 或 HTTPS 协议转为 WebSocket。因此，使用 WebSocket 的应用程序将始终以 HTTP/S 开始，然后进行升级。这种升级发生在什么时候取决于具体的应用;可以在应用启动的时候，或者当一个特定的 URL 被请求的时候。

在我们的应用中，仅当 URL 请求以“/ws”结束时，我们才升级协议为WebSocket。否则，服务器将使用基本的 HTTP/S。一旦连接升级，之后的数据传输都将使用 WebSocket 。

下面看下服务器的逻辑图

Figure 11.2 Server logic



#1客户端/用户连接到服务器并加入聊天

#2 HTTP 请求页面或 WebSocket 升级握手

#3服务器处理所有客户端/用户

#4响应 URI “/”的请求，转到 index.html

#5如果访问的是 URI“/ws” ，处理 WebSocket 升级握手

#6升级握手完成后 ，通过 WebSocket 发送聊天消息

处理 HTTP 请求

本节我们将实现此应用中用于处理 HTTP 请求的组件，这个组件托管着可供客户端访问的聊天室页面，并且显示客户端发送的消息。

下面就是这个 HttpRequestHandler 的代码,它是一个用来处理 FullHttpRequest 消息的 ChannelInboundHandler 的实现类。注意看它是怎么实现忽略符合 "/ws" 格式的 URI 请求的。

Listing 11.1 HTTPRequestHandler

```
public class HttpRequestHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
    //1
    private final String wsUri;
    private static final File INDEX;

    static {
        URL location = HttpRequestHandler.class.getProtectionDomain().getCodeSource().
getLocation();
        try {
            String path = location.toURI() + "index.html";
            path = !path.contains("file:") ? path : path.substring(5);
            INDEX = new File(path);
        } catch (URISyntaxException e) {
            throw new IllegalStateException("Unable to locate index.html", e);
        }
    }

    public HttpRequestHandler(String wsUri) {
        this.wsUri = wsUri;
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest request) throw
s Exception {
        if (wsUri.equalsIgnoreCase(request.getUri())) {
            ctx.fireChannelRead(request.retain());                    //2
        } else {
            if (HttpHeaders.is100ContinueExpected(request)) {
                send100Continue(ctx);                                //3
            }

            RandomAccessFile file = new RandomAccessFile(INDEX, "r");//4
```

```
            HttpResponse response = new DefaultHttpResponse(request.getProtocolVersion
(), HttpResponseStatus.OK);
            response.headers().set(HttpHeaders.Names.CONTENT_TYPE, "text/html; charset
=UTF-8");

            boolean keepAlive = HttpHeaders.isKeepAlive(request);

            if (keepAlive) {                                        //5
                response.headers().set(HttpHeaders.Names.CONTENT_LENGTH, file.length()
);
                response.headers().set(HttpHeaders.Names.CONNECTION, HttpHeaders.Value
s.KEEP_ALIVE);
            }
            ctx.write(response);                    //6

            if (ctx.pipeline().get(SslHandler.class) == null) {        //7
                ctx.write(new DefaultFileRegion(file.getChannel(), 0, file.length()));
            } else {
                ctx.write(new ChunkedNioFile(file.getChannel()));
            }
            ChannelFuture future = ctx.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTEN
T);           //8
            if (!keepAlive) {
                future.addListener(ChannelFutureListener.CLOSE);        //9
            }
        }
    }

    private static void send100Continue(ChannelHandlerContext ctx) {
        FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1,
HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
            throws Exception {
        cause.printStackTrace();
        ctx.close();
    }
}
```

1.扩展 SimpleChannelInboundHandler 用于处理 FullHttpRequest信息

2.如果请求是一次升级了的 WebSocket 请求，则递增引用计数器（retain）并且将它传递给在 ChannelPipeline 中的下个 ChannelInboundHandler

3.处理符合 HTTP 1.1的 "100 Continue" 请求

4.读取 index.html

5.判断 keepalive 是否在请求头里面

6.写 HttpResponse 到客户端

7.写 index.html 到客户端，根据 ChannelPipeline 中是否有 SslHandler 来决定使用 DefaultFileRegion 还是 ChunkedNioFile

8.写并刷新 LastHttpContent 到客户端，标记响应完成

9.如果 请求头中不包含 keepalive，当写完成时，关闭 Channel

HttpRequestHandler 做了下面几件事，

- 如果该 HTTP 请求被发送到URI "/ws"，则调用 FullHttpRequest 上的 retain()，并通过调用 fireChannelRead(msg) 转发到下一个 ChannelInboundHandler。retain() 的调用是必要的，因为 channelRead() 完成后，它会调用 FullHttpRequest 上的 release() 来释放其资源。（请参考我们先前在第6章中关于 SimpleChannelInboundHandler 的讨论）
- 如果客户端发送的 HTTP 1.1 头是"Expect: 100-continue" ，则发送"100 Continue"的响应。
- 在 头被设置后，写一个 HttpResponse 返回给客户端。注意，这不是 FullHttpResponse，这只是响应的第一部分。另外，这里我们也不使用 writeAndFlush()， 这个是在留在最后完成。
- 如果传输过程既没有要求加密也没有要求压缩，那么把 index.html 的内容存储在一个 DefaultFileRegion 里就可以达到最好的效率。这将利用零拷贝来执行传输。出于这个原因，我们要检查 ChannelPipeline 中是否有一个 SslHandler。如果是的话，我们就使用 ChunkedNioFile。
- 写 LastHttpContent 来标记响应的结束，并终止它
- 如果不要求 keepalive ，添加 ChannelFutureListener 到 ChannelFuture 对象的最后写入，并关闭连接。注意，这里我们调用 writeAndFlush() 来刷新所有以前写的信息。

这里展示了应用程序的第一部分，用来处理纯的 HTTP 请求和响应。接下来我们将处理 WebSocket 的 frame（帧），用来发送聊天消息。

WebSocket frame

WebSockets 在"帧"里面来发送数据，其中每一个都代表了一个消息的一部分。一个完整的消息可以利用了多个帧。

处理 **WebSocket frame**

WebSocket "Request for Comments" (RFC) 定义了六种不同的 frame; Netty 给他们每个都提供了一个 POJO 实现 ，见下表：

Table 11.1 WebSocketFrame types

名称	描述
BinaryWebSocketFrame	contains binary data
TextWebSocketFrame	contains text data
ContinuationWebSocketFrame	contains text or binary data that belongs to a previous BinaryWebSocketFrame or TextWebSocketFrame
CloseWebSocketFrame	represents a CLOSE request and contains close status code and a phrase
PingWebSocketFrame	requests the transmission of a PongWebSocketFrame
PongWebSocketFrame	sent as a response to a PingWebSocketFrame

我们的程序只需要使用下面4个帧类型：

- CloseWebSocketFrame
- PingWebSocketFrame
- PongWebSocketFrame
- TextWebSocketFrame

在这里我们只需要处理 TextWebSocketFrame，其他的会由 WebSocketServerProtocolHandler 自动处理。

下面代码展示了 ChannelInboundHandler 处理 TextWebSocketFrame，同时也将跟踪在 ChannelGroup 中所有活动的 WebSocket 连接

Listing 11.2 Handles Text frames

```
public class TextWebSocketFrameHandler extends SimpleChannelInboundHandler<TextWebSock
etFrame> { //1
    private final ChannelGroup group;

    public TextWebSocketFrameHandler(ChannelGroup group) {
        this.group = group;
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Excep
tion {    //2
        if (evt == WebSocketServerProtocolHandler.ServerHandshakeStateEvent.HANDSHAKE_
COMPLETE) {

            ctx.pipeline().remove(HttpRequestHandler.class);    //3

            group.writeAndFlush(new TextWebSocketFrame("Client " + ctx.channel() + " j
oined"));//4

            group.add(ctx.channel());    //5
        } else {
            super.userEventTriggered(ctx, evt);
        }
    }

    @Override
    public void channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) throws
 Exception {
        group.writeAndFlush(msg.retain());    //6
    }
}
```

1.扩展 SimpleChannelInboundHandler 用于处理 TextWebSocketFrame 信息

2.覆写userEventTriggered() 方法来处理自定义事件

3.如果接收的事件表明握手成功,就从 ChannelPipeline 中删除HttpRequestHandler ，因为接下来不会接受 HTTP 消息了

4.写一条消息给所有的已连接 WebSocket 客户端，通知它们建立了一个新的 Channel 连接

5.添加新连接的 WebSocket Channel 到 ChannelGroup 中，这样它就能收到所有的信息

6.保留收到的消息，并通过 writeAndFlush() 传递给所有连接的客户端。

上面显示了 TextWebSocketFrameHandler 仅作了几件事：

- 当WebSocket 与新客户端已成功握手完成，通过写入信息到 ChannelGroup 中的 Channel 来通知所有连接的客户端，然后添加新 Channel 到 ChannelGroup
- 如果接收到 TextWebSocketFrame，调用 retain() ，并将其写、刷新到 ChannelGroup，

> 使所有连接的 WebSocket Channel 都能接收到它。和以前一样，retain() 是必需的，因为当 channelRead0（）返回时，TextWebSocketFrame 的引用计数将递减。由于所有操作都是异步的，writeAndFlush() 可能会在以后完成，我们不希望它访问无效的引用。

由于 Netty 在其内部处理了其余大部分功能，唯一剩下的需要我们去做的就是为每一个新创建的 Channel 初始化 ChannelPipeline 。要完成这个，我们需要一个ChannelInitializer

初始化 ChannelPipeline

接下来，我们需要安装我们上面实现的两个 ChannelHandler 到 ChannelPipeline。为此，我们需要继承 ChannelInitializer 并且实现 initChannel()。看下面 ChatServerInitializer 的代码实现

Listing 11.3 Init the ChannelPipeline

```
public class ChatServerInitializer extends ChannelInitializer<Channel> {    //1
    private final ChannelGroup group;

    public ChatServerInitializer(ChannelGroup group) {
        this.group = group;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {            //2
        ChannelPipeline pipeline = ch.pipeline();
        pipeline.addLast(new HttpServerCodec());
        pipeline.addLast(new HttpObjectAggregator(64 * 1024));
        pipeline.addLast(new ChunkedWriteHandler());
        pipeline.addLast(new HttpRequestHandler("/ws"));
        pipeline.addLast(new WebSocketServerProtocolHandler("/ws"));
        pipeline.addLast(new TextWebSocketFrameHandler(group));
    }
}
```

1.扩展 ChannelInitializer

2.添加 ChannelHandler　到 ChannelPipeline

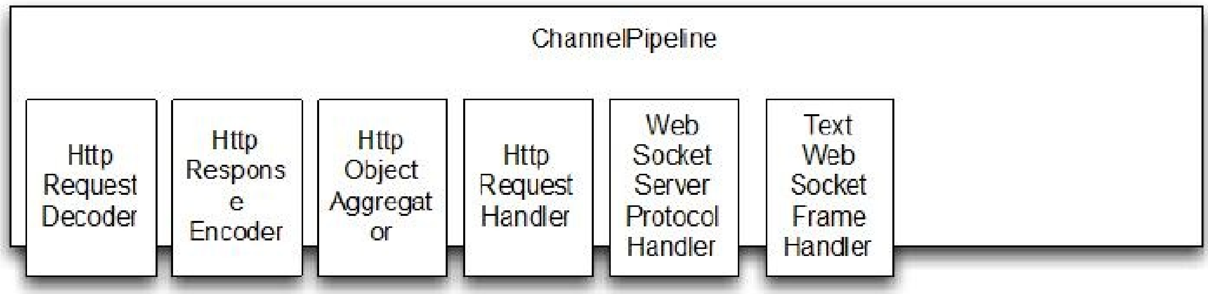
initChannel() 方法用于设置所有新注册的 Channel 的ChannelPipeline,安装所有需要的 ChannelHandler。总结如下：

Table 11.2 ChannelHandlers for the WebSockets Chat server

ChannelHandler	职责
HttpServerCodec	Decode bytes to HttpRequest, HttpContent, LastHttpContent.Encode HttpRequest, HttpContent, LastHttpContent to bytes.
ChunkedWriteHandler	Write the contents of a file.
HttpObjectAggregator	This ChannelHandler aggregates an HttpMessage and its following HttpContents into a single FullHttpRequest or FullHttpResponse (depending on whether it is being used to handle requests or responses).With this installed the next ChannelHandler in the pipeline will receive only full HTTP requests.
HttpRequestHandler	Handle FullHttpRequests (those not sent to "/ws" URI).
WebSocketServerProtocolHandler	As required by the WebSockets specification, handle the WebSocket Upgrade handshake, PingWebSocketFrames,PongWebSocketFrames and CloseWebSocketFrames.
TextWebSocketFrameHandler	Handles TextWebSocketFrames and handshake completion events

该 WebSocketServerProtocolHandler 处理所有规定的 WebSocket 帧类型和升级握手本身。如果握手成功所需的 ChannelHandler 被添加到管道，而那些不再需要的则被去除。管道升级之前的状态如下图。这代表了 ChannelPipeline 刚刚经过 ChatServerInitializer 初始化。

Figure 11.3 ChannelPipeline before WebSockets Upgrade

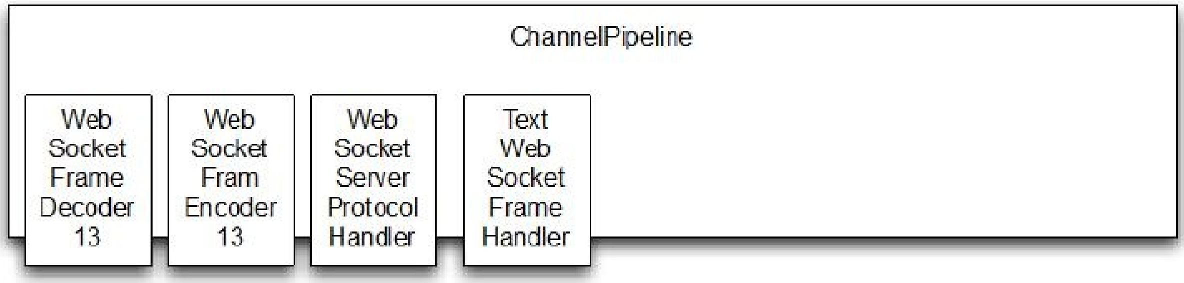


握手升级成功后 WebSocketServerProtocolHandler 替换HttpRequestDecoder 为 WebSocketFrameDecoder，HttpResponseEncoder 为WebSocketFrameEncoder。 为了最大化性能，WebSocket 连接不需要的 ChannelHandler 将会被移除。其中就包括了 HttpObjectAggregator 和 HttpRequestHandler

下图，展示了 ChannelPipeline 经过这个操作完成后的情况。注意 Netty 目前支持四个版本 WebSocket 协议，每个通过其自身的方式实现类。选择正确的版本 WebSocketFrameDecoder 和 WebSocketFrameEncoder 是自动进行的，这取决于在客户端

（在这里指浏览器）的支持（在这个例子中，我们假设使用版本是 13 的 WebSocket 协议，从而图中显示的是 WebSocketFrameDecoder13 和 WebSocketFrameEncoder13）。

Figure 11.4 ChannelPipeline after WebSockets Upgrade



引导

最后一步是 引导服务器，设置 ChannelInitializer

```
public class ChatServer {

    private final ChannelGroup channelGroup = new DefaultChannelGroup(ImmediateEventEx
ecutor.INSTANCE);//1
    private final EventLoopGroup group = new NioEventLoopGroup();
    private Channel channel;

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap(); //2
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
                .childHandler(createInitializer(channelGroup));
        ChannelFuture future = bootstrap.bind(address);
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
   //3
       return new ChatServerInitializer(group);
    }

    public void destroy() {        //4
        if (channel != null) {
            channel.close();
        }
        channelGroup.close();
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception{
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        final ChatServer endpoint = new ChatServer();
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

1.创建 DefaultChannelGroup 用来 保存所有连接的的 WebSocket channel

2.引导 服务器

3.创建 ChannelInitializer

4.处理服务器关闭，包括释放所有资源

测试程序

使用下面命令启动服务器：

```
mvn -PChatServer clean package exec:exec
```

其中项目中的 pom.xml 是配置了 9999 端口。你也可以通过下面的方法修改属性

```
mvn -PChatServer -Dport=1111 clean package exec:exec
```

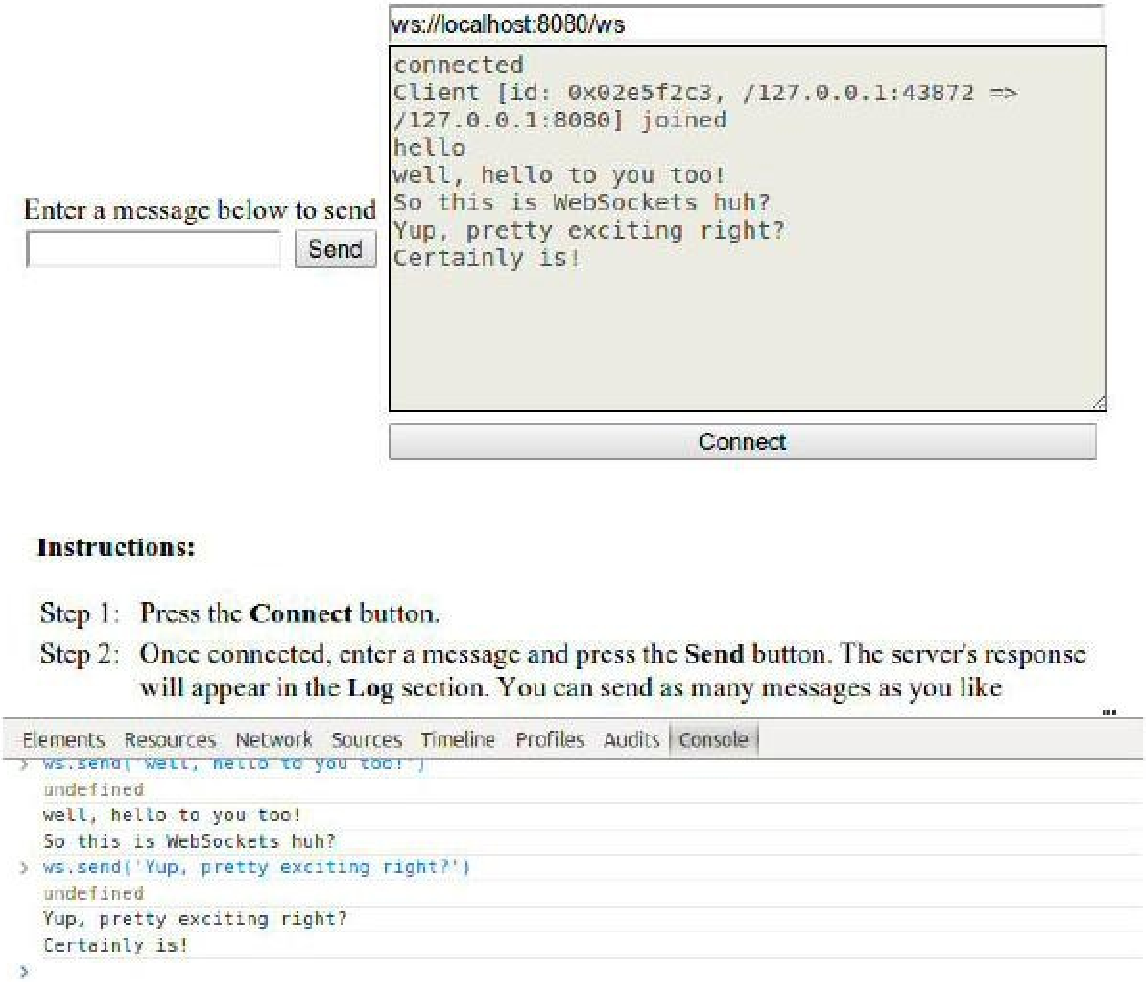
下面是控制台的主要输出(删除了部分行)

Listing 11.5 Compile and start the ChatServer

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: D:/netty-in-action/chapter11/target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting ChatServer on port 9999
```

可以在浏览器中通过 http://localhost:9999 地址访问程序。图11.5展示了此程序在Chrome浏览器下的用户界面。

Figure 11.5 WebSockets ChatServer demonstration



图中显示了两个已经连接了的客户端。第一个客户端是通过上面的图形界面连接的，第二个是通过Chrome浏览器底部的命令行连接的。 你可以注意到，这两个客户端都在发送消息，每条消息都会显示在两个客户端上。

如何加密？

在实际场景中，加密是必不可少的。在Netty中实现加密并不麻烦，你只需要向 ChannelPipeline 中添加 SslHandler ，然后配置一下即可。如下：

Listing 11.6 Add encryption to the ChannelPipeline

```
public class SecureChatServerIntializer extends ChatServerInitializer {    //1
    private final SslContext context;

    public SecureChatServerIntializer(ChannelGroup group, SslContext context) {
        super(group);
        this.context = context;
    }

    @Override
    protected void initChannel(Channel ch) throws Exception {
        super.initChannel(ch);
        SSLEngine engine = context.newEngine(ch.alloc());
        engine.setUseClientMode(false);
        ch.pipeline().addFirst(new SslHandler(engine)); //2
    }
}
```

1.扩展 ChatServerInitializer 来实现加密

2.向 ChannelPipeline 中添加SslHandler

最后修改 ChatServer，使用 SecureChatServerInitializer 并传入 SSLContext

Listing 11.7 Add encryption to the ChatServer

```
public class SecureChatServer extends ChatServer {//1

    private final SslContext context;

    public SecureChatServer(SslContext context) {
        this.context = context;
    }

    @Override
    protected ChannelInitializer<Channel> createInitializer(ChannelGroup group) {
        return new SecureChatServerIntializer(group, context);    //2
    }

    public static void main(String[] args) throws Exception{
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);
        SelfSignedCertificate cert = new SelfSignedCertificate();
        SslContext context = SslContext.newServerContext(cert.certificate(), cert.priv
ateKey());
        final SecureChatServer endpoint = new SecureChatServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

1.扩展 ChatServer

2.返回先前创建的 SecureChatServerInitializer 来启用加密

这样，就在所有的通信中使用了 SSL/TLS 加密。和前面一样，你可以使用Maven拉取应用需要的所有依赖，并启动它，如下所示。

Listing 11.8 Start the SecureChatServer

```
$ mvn -PSecureChatServer clean package exec:exec
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building ChatServer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: D:/netty-in-action/chapter11/target/chat-server-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ chat-server ---
Starting SecureChatServer on port 9999
```

现在你可以通过 HTTPS 地址: https://localhost:9999 来访问SecureChatServer 了。

总结

在本章中，我们学习了如何使用 Netty 中的 WebSocket 来管理 Web 应用程序中的实时数据。我们讲了所支持的数据类型，并讨论了你可能会遇到的问题。虽然 WebSockets 并不能在所有情况下使用，但应该清楚，它代表了 web 技术发展上的一个重要进步。

接下来我们来谈谈“Web2.0”开发中的另一项技术。也许你还没有听说过“SPDY”，但只要你读了下一章，你就很可能在你将来的开发中很好的运用这门技术了。

SPDY

本章介绍

- SPDY 总览
- ChannelHandler, Decoder, 和 Encoder
- 引导一个基于 Netty 的应用
- 测试 SPDY/HTTPS

SPDY(读作"speedy")是一个谷歌开发的开放的网络协议，主要运用于 web 内容传输。SPDY 操纵 HTTP 流量,目标是减少 web 页面加载延迟,提高网络安全。SPDY 达到通过压缩、多路复用和优先级来减少延迟，虽然这取决于网络和网站部署条件的组合。"SPDY"这个名字是谷歌的一个商标,不是一个首字母缩写。（摘自 *http://en.wikipedia.org/wiki/SPDY*）

Netty 的包支持 SPDY。正如我们已经看到在其他情况下,这种支持将使您能够使用 SPDY 无需担心所有的内部细节。在这一章里,我们将提供你需要的所有信息关于在您的应用程序中启用 SPDY ，并同时支持 SPDY 和 HTTP。

SPDY 背景

Google 开发 SPDY 是为了解决扩展性的问题。主要的任务是加载内容的速度更快，做了如下工作：

- 每个头都是压缩的，消息体的压缩是可选的,因为它可能对代理服务器有问题
- 所有的加密都使用 TLS 每个连接多个转移是可能的 数据集可以单独设置优先级,使关键内容先被转移

下表是与 HTTP 的对比

Table 12.1 Comparison of SPDY and HTTP

浏览器	**HTTP 1.1**	**SPDY**
加密	Not by default	Yes
Header 压缩	No	Yes
全双工	No	Yes
Server push	No	Yes
优先级	No	Yes

一些使用场合和指标显示，可以 SPDY 让页面加载速度比H TTP 原先快50%。

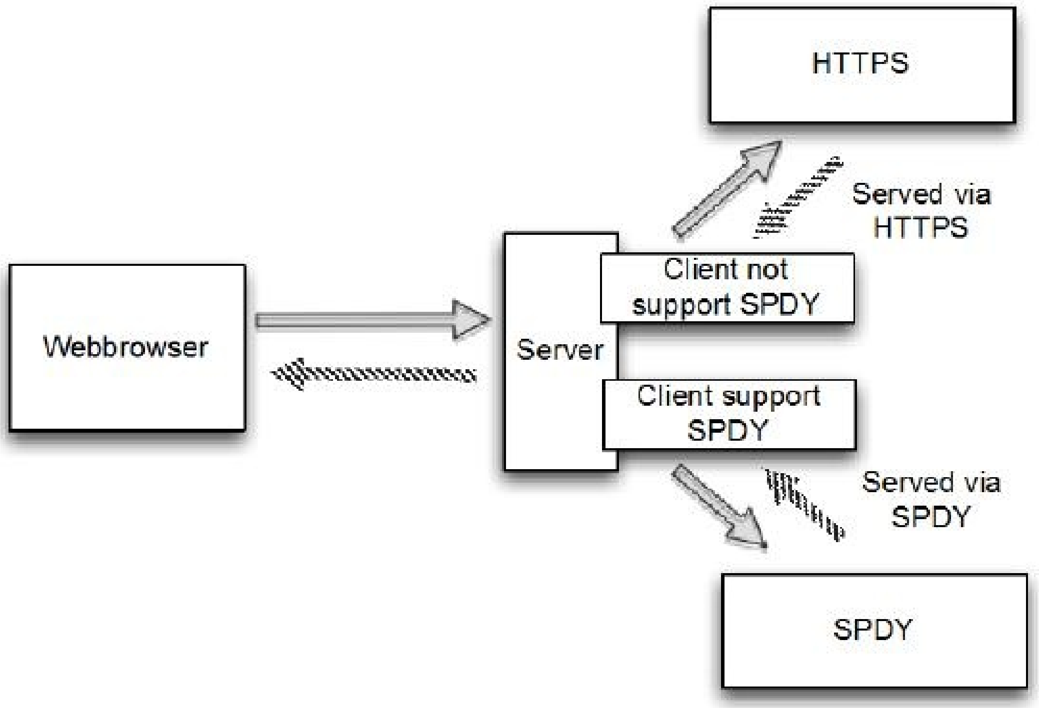
现在 SPDY 的协议草案规范是 1, 2 和 3， Netty 支持 2和3，主要考虑到这个是被广大浏览器所支持的版本。 现在很多浏览器都支持 SPDY，见下表：

Table 12.2 Browsers that support SPDY

浏览器	版本
Chrome	19+
Chromium	19+
Mozilla Firefox	11+ (从 13 起默认开启)
Opera	12.10+

示例程序

编写一个简单的服务器应用程序,向您展示如何将 SPDY 集成到你的下一个应用程序。它只会提供一些静态内容回客户机。这些内容将取决于所使用协议是 HTTPS 或 SPDY 。如果 服务器提供 SPDY 是可以被客户端浏览器所支持，则自动切换到 SPDY 。图12.1显示了应用程序的流程



对于这个应用程序只编写一个服务器组件处理 HTTPS 和 SPDY。为了演示其功能使用两个不同的 web 浏览器,一个支持 SPDY,另外一个不支持。

实现

SPDY 使用 TLS 的扩展称为 Next Protocol Negotiation (NPN)。在Java 中,我们有两种不同的方式选择的基于 NPN 的协议:

* 使用 ssl_npn,NPN 的开源 SSL 提供者。
* 使用通过 Jetty 的 NPN 扩展库。

在这个例子中使用 Jetty 库。如果你想使用 ssl_npn,请参阅https://github.com/benmmurphy/ssl_npn项目文档

Jetty NPN 库

Jetty NPN 库是一个外部的库,而不是 Netty 的本身的一部分。它用于处理 Next Protocol Negotiation, 这是用于检测客户端是否支持 SPDY。

集成 Next Protocol Negotiation

Jetty 库提供了一个接口称为 ServerProvider,确定所使用的协议和选择哪个钩子。这个的实现可能取决于不同版本的 HTTP 和 SPDY 版本的支持。下面的清单显示了将用于我们的示例应用程序的实现。

Listing 12.1 Implementation of ServerProvider

```
public class DefaultServerProvider implements NextProtoNego.ServerProvider {
    private static final List<String> PROTOCOLS =
            Collections.unmodifiableList(Arrays.asList("spdy/2", "spdy/3", "http/1.1")
);  //1

    private String protocol;

    @Override
    public void unsupported() {
        protocol = "http/1.1";   //2
    }

    @Override
    public List<String> protocols() {
        return PROTOCOLS;   //3
    }

    @Override
    public void protocolSelected(String protocol) {
        this.protocol = protocol;  //4
    }

    public String getSelectedProtocol() {
        return protocol;  //5
    }
}
```

1. 定义所有的 ServerProvider 实现的协议
2. 设置如果 SPDY 协议失败了就转到 http/1.1
3. 返回支持的协议的列表
4. 设置选择的协议
5. 返回选择的协议

在 ServerProvider 的实现，我们支持下面的3种协议:

- SPDY 2
- SPDY 3
- HTTP 1.1

如果客户端不支持 SPDY ，则默认使用 HTTP 1.1

实现各种 ChannelHandler

第一个 ChannelInboundHandler 是用于不支持 SPDY 的情况下处理客户端 HTTP 请求，如果不支持 SPDY 就回滚使用默认的 HTTP 协议。

清单12.2显示了HTTP流量的处理程序。

Listing 12.2 Implementation that handles HTTP

```
@ChannelHandler.Sharable
public class HttpRequestHandler extends SimpleChannelInboundHandler<FullHttpRequest> {
    @Override
    public void channelRead0(ChannelHandlerContext ctx, FullHttpRequest request) throw
s Exception { //1
        if (HttpHeaders.is100ContinueExpected(request)) {
            send100Continue(ctx); //2
        }

        FullHttpResponse response = new DefaultFullHttpResponse(request.getProtocolVer
sion(), HttpResponseStatus.OK); //3
        response.content().writeBytes(getContent().getBytes(CharsetUtil.UTF_8));  //4
        response.headers().set(HttpHeaders.Names.CONTENT_TYPE, "text/plain; charset=UT
F-8");  //5

        boolean keepAlive = HttpHeaders.isKeepAlive(request);

        if (keepAlive) {  //6
            response.headers().set(HttpHeaders.Names.CONTENT_LENGTH, response.content(
).readableBytes());
            response.headers().set(HttpHeaders.Names.CONNECTION, HttpHeaders.Values.KE
EP_ALIVE);
        }
        ChannelFuture future = ctx.writeAndFlush(response);  //7

        if (!keepAlive) {
            future.addListener (ChannelFutureListener.CLOSE); //8
        }
    }

    protected String getContent() {  //9
        return "This content is transmitted via HTTP\r\n";
    }

    private static void send100Continue(ChannelHandlerContext ctx) {  //10
        FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1,
HttpResponseStatus.CONTINUE);
        ctx.writeAndFlush(response);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
            throws Exception {  //11
        cause.printStackTrace();
        ctx.close();
    }
}
```

1. 重写 channelRead0() ,可以被所有的接收到的 FullHttpRequest 调用

2. 检查如果接下来的响应是预期的，就写入
3. 新建 FullHttpResponse,用于对请求的响应
4. 生成响应的内容，将它写入 payload
5. 设置头文件，这样客户端就能知道如何与 响应的 payload 交互
6. 检查请求设置是否启用了 keepalive;如果是这样,将标题设置为符合HTTP RFC
7. 写响应给客户端，并获取到 Future 的引用，用于写完成时，获取到通知
8. 如果响应不是 keepalive，在写完成时关闭连接
9. 返回内容作为响应的 payload
10. Helper 方法生成了100 持续的响应，并写回给客户端
11. 若执行阶段抛出异常，则关闭管道

这就是 Netty 处理标准的 HTTP 。你可能需要分别处理特定 URI ,应对不同的状态代码,这取决于资源存在与否,但基本的概念将是相同的。

我们的下一个任务将会提供一个组件来支持 SPDY 作为首选协议。 Netty 提供了简单的处理 SPDY 方法。这些将使您能够重用FullHttpRequest 和 FullHttpResponse 消息，通过 SPDY 透明地接收和发送他们。

HttpRequestHandler 虽然是我们可以重用代码,我们将改变我们的内容写回客户端只是强调协议变化;通常您会返回相同的内容。下面的清单展示了实现,它扩展了先前的 HttpRequestHandler。

Listing 12.3 Implementation that handles SPDY

```
@ChannelHandler.Sharable
public class SpdyRequestHandler extends HttpRequestHandler {   //1
    @Override
    protected String getContent() {
        return "This content is transmitted via SPDY\r\n";  //2
    }
}
```

1. 继承 HttpRequestHandler 这样就能共享相同的逻辑
2. 生产内容写到 payload。这个重写了 HttpRequestHandler 的 getContent() 的实现

SpdyRequestHandler 继承自 HttpRequestHandler,但区别是:写入的内容的 payload 状态的响应是在 SPDY 写的。

我们可以实现两个处理程序逻辑,将选择一个相匹配的协议。然而添加以前写过的处理程序到 ChannelPipeline 是不够的;正确的编解码器还需要补充。它的责任是检测传输字节数,然后使用 FullHttpResponse 和 FullHttpRequest 的抽象进行工作。

Netty 的附带一个基类,完全能做这个。所有您需要做的是实现逻辑选择协议和选择适当的处理程序。

清单12.4显示了实现,它使用 Netty 的提供的抽象基类。

```
public class DefaultSpdyOrHttpChooser extends SpdyOrHttpChooser {

    public DefaultSpdyOrHttpChooser(int maxSpdyContentLength, int maxHttpContentLength
) {
        super(maxSpdyContentLength, maxHttpContentLength);
    }

    @Override
    protected SelectedProtocol getProtocol(SSLEngine engine) {
        DefaultServerProvider provider = (DefaultServerProvider) NextProtoNego.get(eng
ine);  //1
        String protocol = provider.getSelectedProtocol();
        if (protocol == null) {
            return SelectedProtocol.UNKNOWN; //2
        }
        switch (protocol) {
            case "spdy/2":
                return SelectedProtocol.SPDY_2; //3
            case "spdy/3.1":
                return SelectedProtocol.SPDY_3_1; //4
            case "http/1.1":
                return SelectedProtocol.HTTP_1_1; //5
            default:
                return SelectedProtocol.UNKNOWN; //6
        }
    }

    @Override
    protected ChannelInboundHandler createHttpRequestHandlerForHttp() {
        return new HttpRequestHandler(); //7
    }

    @Override
    protected ChannelInboundHandler createHttpRequestHandlerForSpdy() {
        return new SpdyRequestHandler();  //8
    }
}
```

1. 使用 NextProtoNego 用于获取 DefaultServerProvider 的引用, 用于 SSLEngine
2. 协议不能被检测到。一旦字节已经准备好读,检测过程将重新开始。
3. SPDY 2 被检测到
4. SPDY 3 被检测到
5. HTTP 1.1 被检测到
6. 未知协议被检测到
7. 将会被调用给 FullHttpRequest 消息添加处理器。该方法只会在不支持 SPDY 时调用，那么将会使用 HTTPS
8. 将会被调用给 FullHttpRequest 消息添加处理器。该方法在支持 SPDY 时调用

该实现要注意检测正确的协议并设置 ChannelPipeline 。它可以处理SPDY 版本 2、3 和 HTTP 1.1,但可以很容易地修改 SPDY 支持额外的版本。

设置 ChannelPipeline

通过实现 ChannelInitializer 将所有的处理器连接到一起。正如你所了解的那样,这将设置 ChannelPipeline 并添加所有需要的ChannelHandler 的。

SPDY 需要两个 ChannelHandler:

- SslHandler,用于检测 SPDY 是否通过 TLS 扩展
- DefaultSpdyOrHttpChooser,用于当协议被检测到时，添加正确的 ChannelHandler 到 ChannelPipeline

除了添加 ChannelHandler 到 ChannelPipeline, ChannelInitializer 还有另一个责任;即,分配之前创建的 DefaultServerProvider 通过 SslHandler 到 SslEngine 。这将通过Jetty NPN 类库的 NextProtoNego helper 类实现

Listing 12.5 Implementation that handles SPDY

```
public class SpdyChannelInitializer extends ChannelInitializer<SocketChannel> {  //1
    private final SslContext context;

    public SpdyChannelInitializer(SslContext context) //2 {
        this.context = context;
    }

    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        SSLEngine engine = context.newEngine(ch.alloc());  //3
        engine.setUseClientMode(false);  //4

        NextProtoNego.put(engine, new DefaultServerProvider());  //5
        NextProtoNego.debug = true;

        pipeline.addLast("sslHandler", new SslHandler(engine));  //6
        pipeline.addLast("chooser", new DefaultSpdyOrHttpChooser(1024 * 1024, 1024 * 1
024));
    }
}
```

1. 继承 ChannelInitializer 是一个简单的开始
2. 传递 SSLContext 用于创建 SSLEngine
3. 新建 SSLEngine,用于新的管道和连接
4. 配置 SSLEngine 用于非客户端使用
5. 通过 NextProtoNego helper 类绑定 DefaultServerProvider 到 SSLEngine

6. 添加 SslHandler 到 ChannelPipeline 这将会在协议检测到时保存在 ChannelPipeline
7. 添加 DefaultSpyOrHttpChooser 到 ChannelPipeline 。这个实现将会监测协议。添加正确的 ChannelHandler 到 ChannelPipeline,并且移除自身

实际的 ChannelPipeline 设置将会在 DefaultSpdyOrHttpChooser 实现之后完成,因为在这一点上它可能只需要知道客户端是否支持 SPDY

为了说明这一点,让我们总结一下,看看不同 ChannelPipeline 状态期间与客户连接的生命周期。图12.2显示了在 Channel 初始化后的 ChannelPipeline。

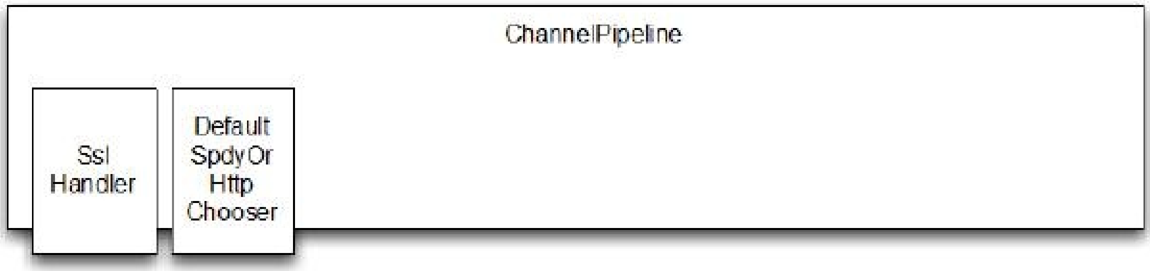


Figure 12.2 ChannelPipeline after connection

现在,这取决于客户端是否支持 SPDY,管道将修改DefaultSpdyOrHttpChooser 来处理协议。之后并不需要添加所需的 ChannelHandler 到 ChannelPipeline,所以删除本身。这个逻辑是由抽象 SpdyOrHttpChooser 类封装,DefaultSpdyOrHttpChooser 父类。

图12.3显示了支持 SPDY 的 ChannelPipeline 用于连接客户端的配置。

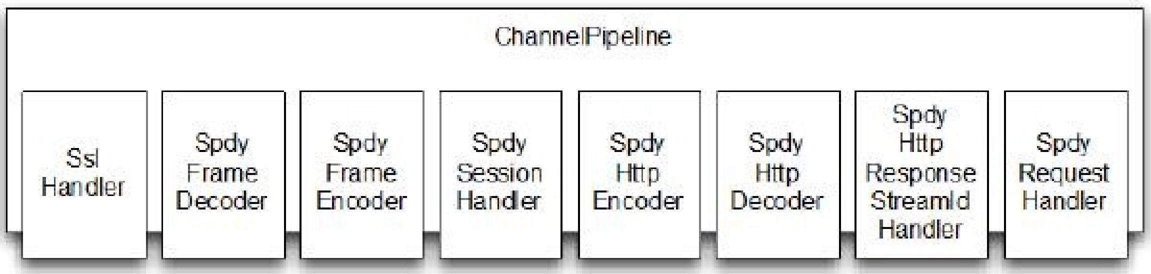


Figure 12.3 ChannelPipeline if SPDY is supported

每个 ChannelHandler 负责的一小部分工作,这个就是对基于 Netty 构造的应用程序最完美的诠释。每个 ChannelHandler 的职责如表12.3所示。

Table 12.3 Responsibilities of the ChannelHandlers when SPDY is used

名称	职责
SslHandler	加解密两端交换的数据
SpdyFrameDecoder	从接收到的 SPDY 帧中解码字节
SpdyFrameEncoder	编码 SPDY 帧到字节
SpdySessionHandler	处理 SPDY session
SpdyHttpEncoder	编码 HTTP 消息到 SPDY 帧
SpdyHttpDecoder	解码 SDPY 帧到 HTTP 消息
SpdyHttpResponseStreamIdHandler	处理基于 SPDY ID 请求和响应之间的映射关系
SpdyRequestHandler	处理 FullHttpRequest, 用于从 SPDY 帧中解码，因此允许 SPDY 透明传输使用

当协议是 HTTP(s) 时，ChannelPipeline 看起来相当不同,如图13.4所示。

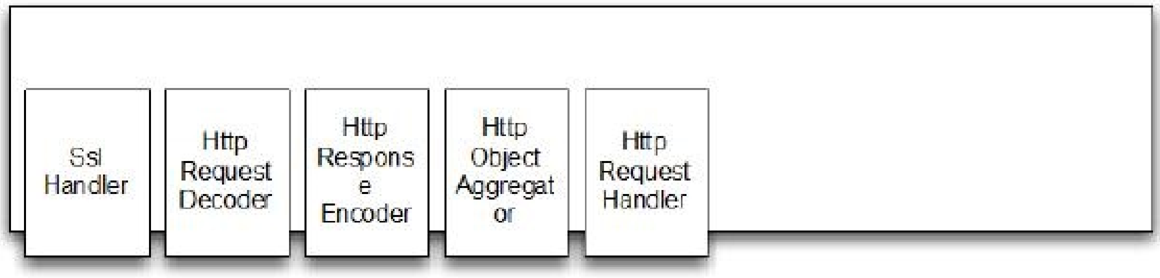


Figure 12.3 ChannelPipeline if SPDY is not supported

和之前一样,每个 ChannelHandler 都有职责,定义在表12.4

Table 12.4 Responsibilities of the ChannelHandlers when HTTP is used

名称	职责
SslHandler	加解密两端交换的数据
HttpRequestDecoder	从接收到的 HTTP 请求中解码字节
HttpResponseEncoder	编码 HTTP 响应到字节

HttpObjectAggregator 处理 SPDY session HttpRequestHandler | 解码时处理 FullHttpRequest

所有东西组合在一起

所有的 ChannelHandler 实现已经准备好，现在组合成一个 SpdyServer

Listing 12.6 SpdyServer implementation

```
public class SpdyServer {

    private final NioEventLoopGroup group = new NioEventLoopGroup();  //1
    private final SslContext context;
    private Channel channel;

    public SpdyServer(SslContext context) { //2
        this.context = context;
    }

    public ChannelFuture start(InetSocketAddress address) {
        ServerBootstrap bootstrap  = new ServerBootstrap(); //3
        bootstrap.group(group)
                .channel(NioServerSocketChannel.class)
                .childHandler(new SpdyChannelInitializer(context)); //4
        ChannelFuture future = bootstrap.bind(address); //5
        future.syncUninterruptibly();
        channel = future.channel();
        return future;
    }

    public void destroy() { //6
        if (channel != null) {
            channel.close();
        }
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            System.err.println("Please give port as argument");
            System.exit(1);
        }
        int port = Integer.parseInt(args[0]);

        SelfSignedCertificate cert = new SelfSignedCertificate();
        SslContext context = SslContext.newServerContext(cert.certificate(), cert.priv
ateKey());  //7
        final SpdyServer endpoint = new SpdyServer(context);
        ChannelFuture future = endpoint.start(new InetSocketAddress(port));

        Runtime.getRuntime().addShutdownHook(new Thread() {
            @Override
            public void run() {
                endpoint.destroy();
            }
        });
        future.channel().closeFuture().syncUninterruptibly();
    }
}
```

1. 构建新的 NioEventLoopGroup 用于处理 I/O
2. 传递 SSLContext 用于加密
3. 新建 ServerBootstrap 用于配置服务器
4. 配置 ServerBootstrap
5. 绑定服务器用于接收指定地址的连接
6. 销毁服务器，用于关闭管道和 NioEventLoopGroup
7. 从 BogusSslContextFactory 获取 SSLContext 。这是一个虚拟实现进行测试。真正的实现将为 SslContext 配置适当的密钥存储库。

启动 SpdyServer 并测试

请注意,当您使用 Jetty NPN 库需要提供它的位置通过 bootclasspath 的 JVM 参数。这一步是必需的,这样才能访问 SslEngine接口。(`-Xbootclasspath` 选项允许您覆盖标准 JDK 附带的实现类)。

下面的清单显示了特殊的参数(`-Xbootclasspath`)使用。

Listing 12.7 SpdyServer implementation

```
java -Xbootclasspath/p:<path_to_npn_boot_jar> ....
```

最简单的方式是使用 Maven 项目管理：

Listing 12.8 Compile and start SpdyServer with Maven

```
$ mvn clean package exec:exec -Pchapter12-SpdyServer
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
...
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-actionprivate
/
target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
```

可以用2个浏览器进行测试，一个支持 SPDY 一个不支持，这里我们用的是 Google Chrome (支持 SPDY) 和 Safari。

浏览器访问 https://127.0.0.1:9999，会显示 SpdyRequestHandler 的处理结果，如下图

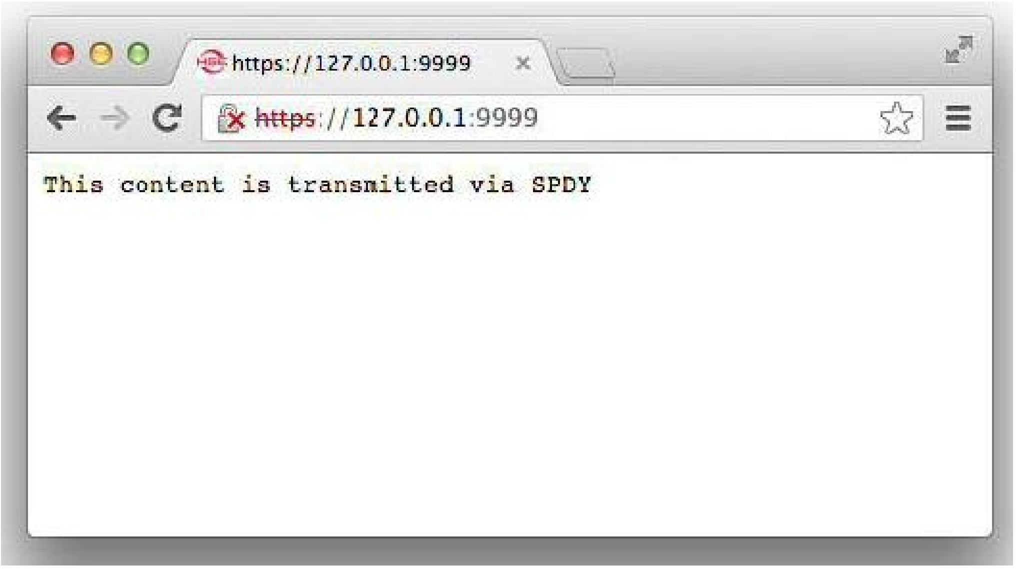


Figure 12.4 SPDY supported by Google Chrome

Google Chrome 的一个很好的功能是可以统计数据，可以很好的看到连接情况。 在浏览器中访问 chrome://net-internals/#spdy 可以看到详细的统计数据

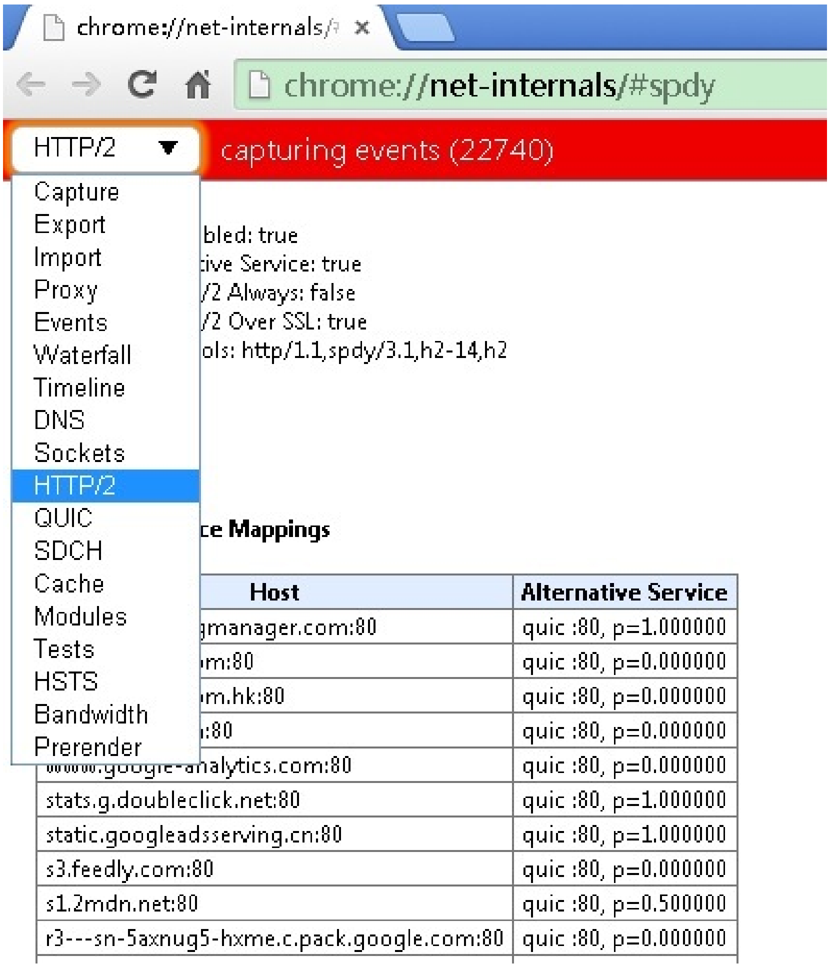


Figure 12.5 SPDY statistics

若不支持 SPDY ，比如我们用 Safari 浏览器访问 https://127.0.0.1:9999 ，则响应将会用 HttpRequestHandler 处理



Figure 12.7 SPDY not supported by Safari

总结

在这一章里,你学习了如何在基于Netty应用程序同时简单的使用 SPDY 和 HTTP(s) 。这提供了一个基础,您可以受益于性能于 SPDY 提供的增强,同时允许现有客户访问您的应用程序。

您学习了如何使用 Netty 提供的 SPDY 助手类,如何使用 Google Chrome获取更多的运行时信息协议。

一路上我们看到了再次修改 ChannelPipeline 如何帮助您构建强大的多路复用器在单个连接的生命周期切换协议。

下一章你学习如何利用高性能、无连接的 UDP。

通过 UDP 广播事件

本章介绍

- UDP 介绍
- ChannelHandler, Decoder, 和 Encoder
- 引导基于 Netty 的应用

前面的章节都是在示例中使用 TCP 协议，这一章，我们将使用UDP。UDP是一种无连接协议，若需要很高的性能和对数据的完成性没有严格要求，那使用 UDP 是一个很好的方法。最著名的基于UDP协议的是用来域名解析的DNS。这一章将给你一个好的理解的无连接协议所以你能够做出明智的决定何时使用 UDP 在您的应用程序。

我们将首先从一个 UDP 的概述,其特点和局限性开始讲解。之后,我们将在本章描述了示例应用程序的开发。

UDP 基础

面向连接的传输协议(如TCP)管理建立一个两个网络端点之间调用(或“连接”),命令和可靠的消息传输在调用的生命周期期间,最后有序在调用终止时终止。与此相反,在这样一个无连接协议 UDP 没有持久连接的概念,每个消息(UDP 数据报)是一个独立的传播。

此外,UDP 没有 TCP 的纠错机制,其中每个对等承认它接收的数据包并由发送方传送包。

以此类推,一个 TCP 连接就像一个电话交谈,一系列的命令消息流在两个方向上。UDP,另一方面,就像把一堆明信片丢进信箱。我们不能知道他们到达目的地的顺序,以及他们是否能够到达。

虽然 UDP 存在某些方面的的局限性,这也解释了为什么它是如此远远快于TCP:所有的握手和消息管理的开销已被消灭。显然,UDP 是一种只适合应用程序可以处理或容忍丢失消息，而不是例如处理金钱交易。

UDP 广播

我们所有的例子这一点利用传输方式称为“单播”:“将消息发送给一个网络拥有唯一地址的目的地”，这种模式支持连接和无连接协议。

然而,UDP 提供了额外的传输模式对多个接收者发送消息:

- 多播:传送给一组主机
- 广播:传送到网络上的所有主机(或子网)

示例应用程序在本章将说明使用 UDP 广播发送消息,可以接收到所有主机在同一网络。为此我们将使用特殊的“有限广播”或“零”网络地址255.255.255.255。消息发送到这个地址是规定要在本地网络(0.0.0.0)的所有主机和从不转发到其他网络通过路由器。

下一节将讨论示例应用程序的设计。

UDP 示例

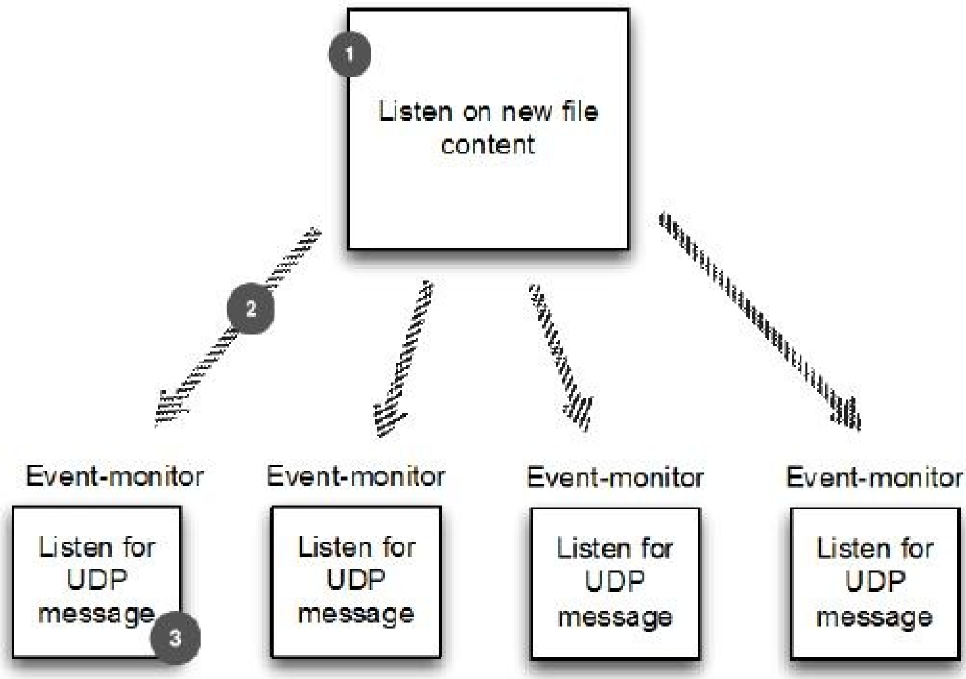
我们的示例应用程序将打开一个文件，将每一行作为消息通过 UDP 发到指定的端口。如果你熟悉类 UNIX 操作系统,可以认为这是一个非常标准的简化版本 “syslog（系统日志）”。“UDP，是一个完美的适合这样的应用程序，因为偶尔丢失一行日志文件可以被容忍,因为文件本身存储在文件系统中。此外,应用程序提供了非常有价值的能力有效地处理大量的数据。

UDP 广播使添加新事件“监视器”接收日志消息一样简单开始一个指定的端口上侦听器程序。然而,这种轻松的访问也提出了一个潜在的安全问题,指出为什么 UD P广播往往是在安全的环境中使用。还要注意广播消息可能只能在本地网络,因为路由器经常阻止他们。

Publish/Subscribe（发布/订阅）

应用程序，如 syslog 通常归类为“发布/订阅”;生产者或服务发布事件和多个订阅者可以收到它们。

整体看下这个应用，如下图：



1. 应用监听新文件内容
2. 事件通过 UDP 广播
3. 事件监视器监听并显示内容

Figure 13.1 Application overview

应用程序有两个组件:广播器和监视器或(可能有多个实例)。为了简单起见我们不会添加身份验证、验证、加密。

在下一节中我们将开始探索实现中,我们还将讨论 UDP 和 TCP 应用程序开发之间的差异。

EventLog 的 POJO

在消息应用里面，数据一般以 POJO 形式呈现。这可能保存配置或处理信息除了实际的消息数据。在这个应用程序里，消息的单元是一个“事件”。由于数据来自一个日志文件，我们将称之为 LogEvent。

清单13.1显示了这个简单的POJO的细节。

Listing 13.1 LogEvent message

```
public final class LogEvent {
    public static final byte SEPARATOR = (byte) ':';

    private final InetSocketAddress source;
    private final String logfile;
    private final String msg;
    private final long received;

    public LogEvent(String logfile, String msg) { //1
        this(null, -1, logfile, msg);
    }

    public LogEvent(InetSocketAddress source, long received, String logfile, String ms
g) {  //2
        this.source = source;
        this.logfile = logfile;
        this.msg = msg;
        this.received = received;
    }

    public InetSocketAddress getSource() { //3
        return source;
    }

    public String getLogfile() { //4
        return logfile;
    }

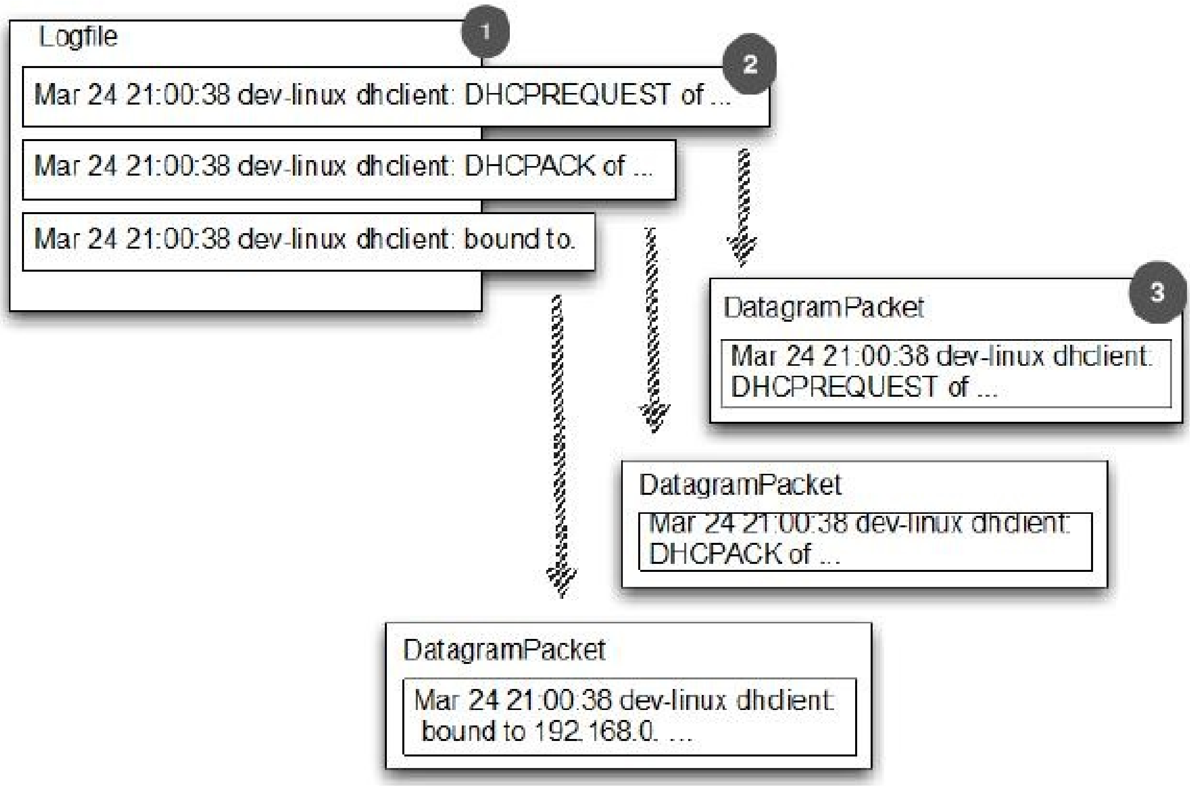
    public String getMsg() {  //5
        return msg;
    }

    public long getReceivedTimestamp() {  //6
        return received;
    }
}
```

1. 构造器用于出站消息
2. 构造器用于入站消息
3. 返回发送 LogEvent 的 InetSocketAddress 的资源
4. 返回用于发送 LogEvent 的日志文件的名称
5. 返回消息的内容
6. 返回 LogEvent 接收到的时间

写广播器

本节，我们将写一个广播器。下图展示了广播一个 DatagramPacket 在每个日志实体里面的方法



1. 日志文件
2. 日志文件中的日志实体
3. 一个 DatagramPacket 保持一个单独的日志实体

Figure 13.2 Log entries sent with DatagramPackets

图13.3表示一个 LogEventBroadcaster 的 ChannelPipeline 的高级视图,说明了 LogEvent 是如何流转的。

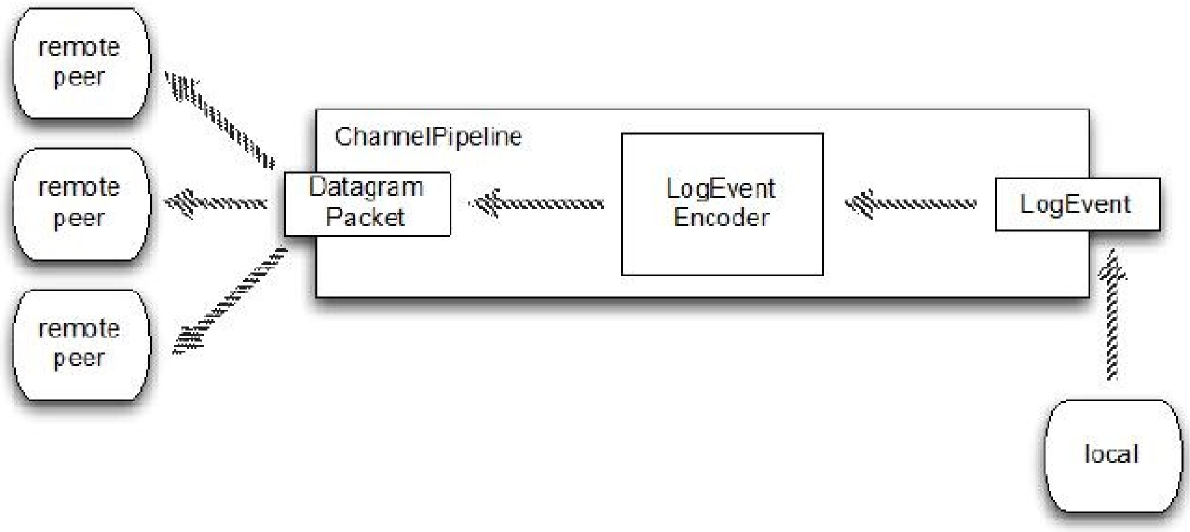


Figure 13.3 LogEventBroadcaster: ChannelPipeline and LogEvent flow

正如我们所看到的,所有的数据传输都封装在 LogEvent 消息里。LogEventBroadcaster 写这些通过在本地端的管道,发送它们通过ChannelPipeline 转换(编码)为一个定制的 ChannelHandler 的DatagramPacket 信息。最后,他们通过 UDP 广播并被远程接收。

编码器和解码器

编码器和解码器将消息从一种格式转换为另一种,深度探讨在第7章中进行。我们探索 *Netty* 提供的基础类来简化和实现自定义 *ChannelHandler* 如 *LogEventEncoder* 在这个应用程序中。

下面展示了 编码器的实现

Listing 13.2 LogEventEncoder

```
public class LogEventEncoder extends MessageToMessageEncoder<LogEvent> {
    private final InetSocketAddress remoteAddress;

    public LogEventEncoder(InetSocketAddress remoteAddress) {  //1
        this.remoteAddress = remoteAddress;
    }

    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext, LogEvent logEve
nt, List<Object> out) throws Exception {
        byte[] file = logEvent.getLogfile().getBytes(CharsetUtil.UTF_8); //2
        byte[] msg = logEvent.getMsg().getBytes(CharsetUtil.UTF_8);
        ByteBuf buf = channelHandlerContext.alloc().buffer(file.length + msg.length +
1);
        buf.writeBytes(file);
        buf.writeByte(LogEvent.SEPARATOR); //3
        buf.writeBytes(msg);  //4
        out.add(new DatagramPacket(buf, remoteAddress));  //5
    }
}
```

1. LogEventEncoder 创建了 DatagramPacket 消息类发送到指定的 InetSocketAddress
2. 写文件名到 ByteBuf
3. 添加一个 SEPARATOR
4. 写一个日志消息到 ByteBuf
5. 添加新的 DatagramPacket 到出站消息

为什么使用 *MessageToMessageEncoder?*

当然我们可以编写自己的自定义 *ChannelOutboundHandler* 来转换 *LogEvent* 对象到 *DatagramPackets*。但是继承自*MessageToMessageEncoder* 为我们简化和做了大部分的工作。

为了实现 LogEventEncoder，我们只需要定义服务器的运行时配置,我们称之为“bootstrapping（引导）”。这包括设置各种 ChannelOption 并安装需要的 ChannelHandler 到 ChannelPipeline 中。完成的 LogEventBroadcaster 类,如清单13.3所示。

Listing 13.3 LogEventBroadcaster

```
public class LogEventBroadcaster {
    private final Bootstrap bootstrap;
    private final File file;
    private final EventLoopGroup group;

    public LogEventBroadcaster(InetSocketAddress address, File file) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
        bootstrap.group(group)
                .channel(NioDatagramChannel.class)
                .option(ChannelOption.SO_BROADCAST, true)
                .handler(new LogEventEncoder(address)); //1

        this.file = file;
    }

    public void run() throws IOException {
        Channel ch = bootstrap.bind(0).syncUninterruptibly().channel(); //2
        System.out.println("LogEventBroadcaster running");
        long pointer = 0;
        for (;;) {
            long len = file.length();
            if (len < pointer) {
                // file was reset
                pointer = len; //3
            } else if (len > pointer) {
                // Content was added
                RandomAccessFile raf = new RandomAccessFile(file, "r");
                raf.seek(pointer);  //4
                String line;
                while ((line = raf.readLine()) != null) {
                    ch.writeAndFlush(new LogEvent(null, -1, file.getAbsolutePath(), li
```

```
ne));  //5
                }
                pointer = raf.getFilePointer(); //6
                raf.close();
            }
            try {
                Thread.sleep(1000);  //7
            } catch (InterruptedException e) {
                Thread.interrupted();
                break;
            }
        }
    }

    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            throw new IllegalArgumentException();
        }

        LogEventBroadcaster broadcaster = new LogEventBroadcaster(new InetSocketAddres
s("255.255.255.255",
                Integer.parseInt(args[0])), new File(args[1]));  //8
        try {
            broadcaster.run();
        } finally {
            broadcaster.stop();
        }
    }
}
```

1. 引导 NioDatagramChannel 。为了使用广播，我们设置 SO_BROADCAST 的 socket 选项
2. 绑定管道。注意当使用 Datagram Channel 时，是没有连接的
3. 如果需要，可以设置文件的指针指向文件的最后字节
4. 设置当前文件的指针，这样不会把旧的发出去
5. 写一个 LogEvent 到管道用于保存文件名和文件实体。(我们期望每个日志实体是一行长度)
6. 存储当前文件的位置，这样，我们可以稍后继续
7. 睡 1 秒。如果其他中断退出循环就重新启动它。
8. 构造一个新的实例 LogEventBroadcaster 并启动它

这就是程序的完整的第一部分。可以使用 "netcat" 程序查看程序的结果。在 UNIX/Linux 系统，可以使用 "nc", 在 Windows 环境下，可以在 http://nmap.org/ncat找到

Netcat 是完美的第一个测试我们的应用程序;它只是监听指定的端口上接收并打印所有数据到标准输出。将其设置为在端口 9999 上监听 UDP 数据如下:

```
$ nc -l -u 9999
```

现在我们需要启动 LogEventBroadcaster。清单13.4显示了如何使用 mvn 编译和运行广播器。pom的配置。pom.xml 配置指向一个文件 `/var/log/syslog` (假设是UNIX / Linux环境)和端口设置为 9999。文件中的条目将通过 UDP 广播到端口,在你开始 netcat 后打印到控制台。

Listing 13.4 Compile and start the LogEventBroadcaster

```
$ mvn clean package exec:exec -Pchapter13-LogEventBroadcaster
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] --------------------------------------------------------------------
...
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-actionprivate
/
target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
LogEventBroadcaster running
```

当调用 mvn 时,在系统属性中改变文件和端口值,指定你想要的。清单13.5 设置日志文件 到 `/var/log/mail.log` 和端口 8888。

Listing 13.5 Compile and start the LogEventBroadcaster

```
$ mvn clean package exec:exec -Pchapter13-LogEventBroadcaster /
-Dlogfile=/var/log/mail.log -Dport=8888 -....
....
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action -
LogEventBroadcaster running
```

当看到 “LogEventBroadcaster running” 说明程序运行成功了。

netcat 只用于测试,但不适合生产环境中使用。

写监视器

这一节我们编写一个监视器：EventLogMonitor ，也就是用来接收事件的程序，用来代替 netcat 。EventLogMonitor 做下面事情：

- 接收 LogEventBroadcaster 广播的 UDP DatagramPacket
- 解码 LogEvent 消息
- 输出 LogEvent 消息

和之前一样,将实现自定义 ChannelHandler 的逻辑。图13.4描述了LogEventMonitor 的 ChannelPipeline 并表明了 LogEvent 的流经情况。

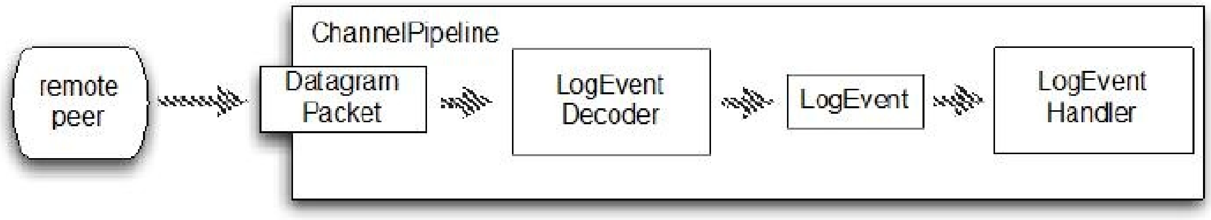


Figure 13.4 LogEventMonitor

图中显示我们的两个自定义 ChannelHandlers,LogEventDecoder 和 LogEventHandler。首先是负责将网络上接收到的 DatagramPacket 解码到 LogEvent 消息。清单13.6显示了实现。

Listing 13.6 LogEventDecoder

```
public class LogEventDecoder extends MessageToMessageDecoder<DatagramPacket> {
    @Override
    protected void decode(ChannelHandlerContext ctx, DatagramPacket datagramPacket, Li
st<Object> out) throws Exception {
        ByteBuf data = datagramPacket.content(); //1
        int i = data.indexOf(0, data.readableBytes(), LogEvent.SEPARATOR);  //2
        String filename = data.slice(0, i).toString(CharsetUtil.UTF_8);  //3
        String logMsg =  data.slice(i + 1, data.readableBytes()).toString(CharsetUtil.
UTF_8);  //4

        LogEvent event = new LogEvent(datagramPacket.recipient(), System.currentTimeMi
llis(),
                filename,logMsg); //5
        out.add(event);
    }
}
```

1. 获取 DatagramPacket 中数据的引用
2. 获取 SEPARATOR 的索引

3. 从数据中读取文件名
4. 读取数据中的日志消息
5. 构造新的 LogEvent 对象并将其添加到列表中

第二个 ChannelHandler 将执行一些首先创建的 LogEvent 消息。在这种情况下,我们只会写入 system.out。在真实的应用程序可能用到一个单独的日志文件或放到数据库。

下面的清单显示了 LogEventHandler。

Listing 13.7 LogEventHandler

```
public class LogEventHandler extends SimpleChannelInboundHandler<LogEvent> { //1

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exc
eption {
        cause.printStackTrace(); //2
        ctx.close();
    }

    @Override
    public void channelRead0(ChannelHandlerContext channelHandlerContext, LogEvent eve
nt) throws Exception {
        StringBuilder builder = new StringBuilder(); //3
        builder.append(event.getReceivedTimestamp());
        builder.append(" [");
        builder.append(event.getSource().toString());
        builder.append("] [");
        builder.append(event.getLogfile());
        builder.append("] : ");
        builder.append(event.getMsg());

        System.out.println(builder.toString()); //4
    }
}
```

1. 继承 SimpleChannelInboundHandler 用于处理 LogEvent 消息
2. 在异常时，输出消息并关闭 channel
3. 建立一个 StringBuilder 并构建输出
4. 打印出 LogEvent 的数据

LogEventHandler 打印出 LogEvent 的一个易读的格式,包括以下:

- 收到时间戳以毫秒为单位
- 发送方的 InetSocketAddress,包括IP地址和端口
- LogEvent 生成绝对文件名
- 实际的日志消息,代表在日志文件中一行

现在我们需要安装处理程序到 ChannelPipeline ，如图13.4所示。下一个清单显示了这是如何实现 LogEventMonitor 类的一部分。

Listing 13.8 LogEventMonitor

```
public class LogEventMonitor {

    private final Bootstrap bootstrap;
    private final EventLoopGroup group;
    public LogEventMonitor(InetSocketAddress address) {
        group = new NioEventLoopGroup();
        bootstrap = new Bootstrap();
        bootstrap.group(group)  //1
                .channel(NioDatagramChannel.class)
                .option(ChannelOption.SO_BROADCAST, true)
                .handler(new ChannelInitializer<Channel>() {
                    @Override
                    protected void initChannel(Channel channel) throws Exception {
                        ChannelPipeline pipeline = channel.pipeline();
                        pipeline.addLast(new LogEventDecoder());  //2
                        pipeline.addLast(new LogEventHandler());
                    }
                }).localAddress(address);

    }

    public Channel bind() {
        return bootstrap.bind().syncUninterruptibly().channel();  //3
    }

    public void stop() {
        group.shutdownGracefully();
    }

    public static void main(String[] args) throws Exception {
        if (args.length != 1) {
            throw new IllegalArgumentException("Usage: LogEventMonitor <port>");
        }
        LogEventMonitor monitor = new LogEventMonitor(new InetSocketAddress(Integer.pa
rseInt(args[0])));  //4
        try {
            Channel channel = monitor.bind();
            System.out.println("LogEventMonitor running");

            channel.closeFuture().await();
        } finally {
            monitor.stop();
        }
    }
}
```

1. 引导 NioDatagramChannel。设置 SO_BROADCAST socket 选项。
2. 添加 ChannelHandler 到 ChannelPipeline
3. 绑定的通道。注意,在使用 DatagramChannel 是没有连接，因为这些 无连接
4. 构建一个新的 LogEventMonitor

运行 LogEventBroadcaster 和 LogEventMonitor

如上所述,我们将使用 Maven 来运行应用程序。这一次你需要打开两个控制台窗口给每个项目。用 Ctrl-C 可以停止它。

首先我们将启动 LogEventBroadcaster 如清单13.4所示,除了已经构建项目以下命令即可(使用默认值):

```
$ mvn exec:exec -Pchapter13-LogEventBroadcaster
```

和之前一样,这将通过 UDP 广播日志消息。

现在,在一个新窗口,构建和启动 LogEventMonitor 接收和显示广播消息。

Listing 13.9 Compile and start the LogEventBroadcaster

```
$ mvn clean package exec:exec -Pchapter13-LogEventMonitor
[INFO] Scanning for projects...
[INFO]
[INFO] --------------------------------------------------------------------
[INFO] Building netty-in-action 0.1-SNAPSHOT
[INFO] --------------------------------------------------------------------
...
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ netty-in-action ---
[INFO] Building jar: /Users/norman/Documents/workspace-intellij/netty-in-actionprivate
/
target/netty-in-action-0.1-SNAPSHOT.jar
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:exec (default-cli) @ netty-in-action ---
LogEventMonitor running
```

当看到 “LogEventMonitor running” 说明程序运行成功了。

控制台将显示任何事件被添加到日志文件中,如下所示。消息的格式是由LogEventHandler 创建。

Listing 13.10 LogEventMonitor output

```
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:55:08 dev-linux
dhclient: bound to 192.168.0.50 -- renewal in 270 seconds.
1364217299382 [/192.168.0.38:63182] [[/var/log/messages] : Mar 25 13:59:38 dev-linux
dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299382 [/192.168.0.38:63182] [/[/var/log/messages] : Mar 25 13:59:38 dev-linux
dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299382 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 13:59:38 dev-linux
dhclient: bound to 192.168.0.50 -- renewal in 259 seconds.
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
dhclient: DHCPREQUEST of 192.168.0.50 on eth2 to 192.168.0.254 port 67
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
dhclient: DHCPACK of 192.168.0.50 from 192.168.0.254
1364217299383 [/192.168.0.38:63182] [/var/log/messages] : Mar 25 14:03:57 dev-linux
dhclient: bound to 192.168.0.50 -- renewal in 285 seconds.
```

若你没有访问 UNIX syslog 的权限，可以创建 自定义的文件，手动填入内容。下面是 UNIX 命令用 touch 创建一个空文件

```
$ touch ~/mylog.log
```

再次启动 LogEventBroadcaster，设置系统属性

```
$ mvn exec:exec -Pchapter13-LogEventBroadcaster -Dlogfile=~/mylog.log
```

当 LogEventBroadcaster 运行时，你可以手动的添加消息到文件来查看广播到 LogEventMonitor 控制台的内容。使用 echo 和输出的文件

```
$ echo 'Test log entry' >> ~/mylog.log
```

你可以启动任意个监视器实例，他们都会收到相同的消息。

总结

本章提供了一个无连接的传输协议，如UDP的介绍。我们看到,在 Netty的您可以从 TCP 切换到 UDP 的同时使用相同的 API。您还了解了如何通过专门的 ChannelHandler 来组织处理逻辑。我们通过独立的解码器的逻辑来处理消息对象。

在下一章中我们将探讨用 Netty 实现可重用的编解码器。

实现自定义的编解码器

本章介绍：

- Decoder
- Encoder
- 单元测试

本章讲述 Netty 中如何轻松实现定制的编解码器，由于 Netty 架构的灵活性，这些编解码器易于重用和测试。为了更容易实现，使用 Memcached 作为协议例子是因为它更方便我们实现。

Memcached 是来自 Memcached.org 的免费开源、高性能、分布式的内存对象缓存系统，其目的是加速动态 Web 应用程序的响应，减轻数据库负载；Memcache 实际上是一个以 key-value 存储任意数据的内存小块。可能有人会问"为什么使用 Memcached？"，因为 Memcached 协议非常简单，便于讲解。

编解码器的范围

我们将只实现 Memcached 协议的一个子集，这足够我们进行添加、检索、删除对象；在 Memcached 中是通过执行 SET,GET,DELETE 命令来实现的。Memcached 支持很多其他的命令，但我们只使用其中三个命令，简单的东西，我们才会理解的更清楚。

Memcached 有一个二进制和纯文本协议，它们都可以用来与 Memcached 服务器通信，使用什么类型的协议取决于服务器支持哪些协议。本章主要关注实现二进制协议，因为二进制在网络编程中最常用。

实现 Memcached 编解码器

当想要实现一个给定协议的编解码器，我们应该花一些事件来了解它的运作原理。通常情况下，协议本身都有一些详细的记录。在这里你会发现多少细节？幸运的是 Memcached 的二进制协议可以很好的扩展。
在 RFC 中有相应的规范，可以在
https://code.google.com/p/Memcached/wiki/MemcacheBinaryProtocol 找到 。

我们不会实现 Memcached 的所有命令，只会实现三种操作：SET,GET 和 DELETE。这样做事为了让事情变得简单。

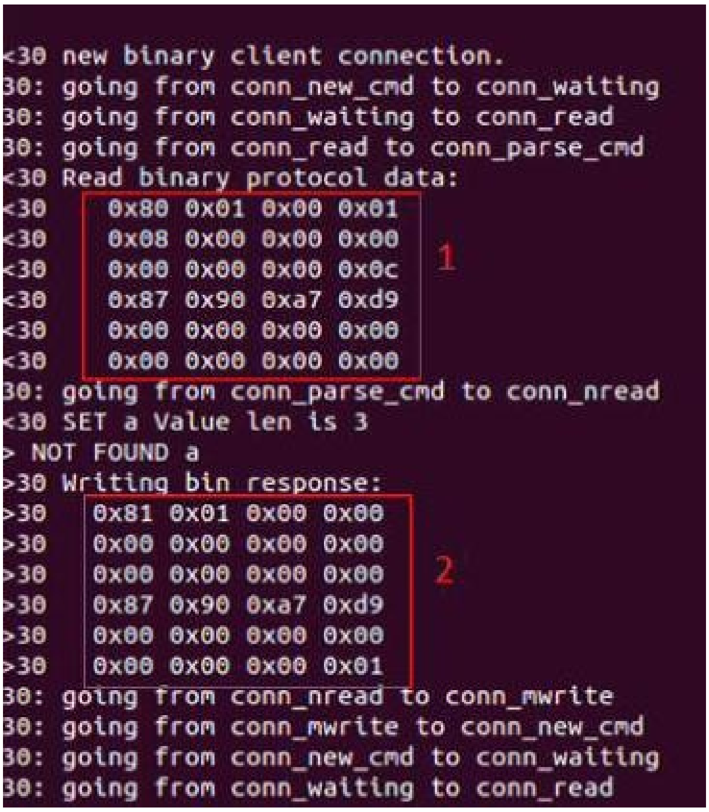
了解 Memcached 二进制协议

我们说,要实现 Memcached 的 GET, SET, 和 DELETE 操作。我们仅仅关注这些,但 memcached 协议有一个通用的结构,只有少数参数改变为了改变一个请求或响应的意义。这意味着您可以轻松地扩展实现添加其他命令。一般协议有 24 字节头用于请求和响应。这个头可以分解如下表14.1中。

Table 14.1 Sample Memcached header byte structure

Field	Byte offset	Value
Magic	0	0x80 用于请求 0x81 用于响应
OpCode	1	0x01...0x1A
Key length	2 和 3	1...32,767
Extra length	4	0x00, x04, 或 0x08
Data type	5	0x00
Reserved	6 和 7	0x00
Total body length	8-11	所有 body 的长度
Opaque	12-15	任何带带符号的 32-bit 整数; 这个也包含在响应中，因此更容易将请求映射到响应。
CAS	16-23	数据版本检查

注意每个部分使用的字节数。这告诉你接下来你应该用什么数据类型。例如,如果字节的偏移量只是 byte 0,那么旧使用一个 Java byte来表示它;如果它是6和7(2字节),你使用一个Java short;如果它是 12-15(4字节),你使用一个Java int,等等。



1. 请求（只有显示头）
2. 响应

Figure 14.2 Real-world Memcached request and response headers

在图14.2中,高亮显示的第一部分代表请求打到 Memcached (只显示请求头),在这种情况下是告诉 Memcached 来 SET 键是“a”而值是“abc”。第部分是响应。

突出显示的部分中的每一行代表4个字节;因为有6行,这意味着请求头是由24个字节,正如我们之前说的。回顾表14.1中,您可以头在一个眞正的请求中看到头文件中的信息。现在,这是所有你需要知道的关于 Memcached 二进制协议。在下一节中,我们需要看看多么我们可以开始制作 Netty 这些请求。

Netty 编码器和解码器

Netty 的是一个复杂和先进的框架,但它并不玄幻。当我们请求一些设置了 key 的给定值时,我们知道 Request 类的一个实例被创建来代表这个请求。但 Netty 并不知道 Request 对象是如何转成 Memcached 所期望的。Memcached 所期望的是字节序列；忽略使用的协议，数据在网络上传输永远是字节序列。

将 Request 对象转为 Memcached 所需的字节序列，Netty 需要用 MemcachedRequest 来编码成另外一种格式。这里所说的另外一种格式不单单是从对象转为字节，也可以是从对象转为对象，或者是从对象转为字符串等。编码器的内容可以详见第七章。

Netty 提供了一个抽象类称为 MessageToByteEncoder。它提供了一个抽象方法,将一条消息(在本例中我们 MemcachedRequest 对象)转为字节。你显示什么信息实现通过使用 Java 泛型可以处理;例如 , MessageToByteEncoder 说这个编码器要编码的对象类型是 MemcachedRequest

MessageToByteEncoder 和 *Java* 泛型

使用 *MessageToByteEncoder* 可以绑定特定的参数类型。如果你有多个不同的消息类型，在相同的编码器里，也可以使用*MessageToByteEncoder*，注意检查消息的类型即可

这也适用于解码器,除了解码器将一系列字节转换回一个对象。 这个 Netty 的提供了 ByteToMessageDecoder 类,而不是提供一个编码方法用来实现解码。在接下来的两个部分你看看如何实现一个 Memcached 解码器和编码器。在你做之前,应该意识到在使用 Netty 时，你不总是需要自己提供编码器和解码器。自所以现在这么做是因为 Netty 没有对 Memcached 内置支持。而 HTTP 以及其他标准的协议，Netty 已经是提供的了。

编码器和解码器

记住,编码器处理出站，而解码器处理入站。这基本上意味着编码器将编码数据,写入远端。解码器将从远端读取处理数据。重要的是要记住,出站和入站是两个不同的方向。

请注意,为了程序简单，我们的编码器和解码器不检查任何值的最大大小。在实际实现中你需要一些验证检查,如果检测到违反协议，则使用 EncoderException 或 DecoderException(或一个子类)。

实现 **Memcached** 编码器

本节我们将简要介绍编码器的实现。正如我们提到的,编码器负责编码消息为字节序列。这些字节可以通过网络发送到远端。为了发送请求，我们首先创建 MemcachedRequest 类,稍后编码器实现会编码为一系列字节。下面的清单显示了我们的 MemcachedRequest 类

Listing 14.1 Implementation of a Memcached request

```
public class MemcachedRequest { //1
    private static final Random rand = new Random();
    private final int magic = 0x80;//fixed so hard coded
    private final byte opCode; //the operation e.g. set or get
    private final String key; //the key to delete, get or set
    private final int flags = 0xdeadbeef; //random
    private final int expires; //0 = item never expires
    private final String body; //if opCode is set, the value
    private final int id = rand.nextInt(); //Opaque
    private final long cas = 0; //data version check...not used
    private final boolean hasExtras; //not all ops have extras

    public MemcachedRequest(byte opcode, String key, String value) {
        this.opCode = opcode;
        this.key = key;
        this.body = value == null ? "" : value;
        this.expires = 0;
        //only set command has extras in our example
        hasExtras = opcode == Opcode.SET;
    }

    public MemcachedRequest(byte opCode, String key) {
        this(opCode, key, null);
    }

    public int magic() { //2
        return magic;
    }

    public int opCode() {  //3
        return opCode;
    }

    public String key() {  //4
        return key;
    }

    public int flags() {  //5
        return flags;
    }

    public int expires() {  //6
        return expires;
    }

    public String body() {  //7
        return body;
    }

    public int id() {  //8
        return id;
    }
```

```
    public long cas() {  //9
        return cas;
    }

    public boolean hasExtras() {  //10
        return hasExtras;
    }
}
```

1. 这个类将会发送请求到 Memcached server
2. 幻数，它可以用来标记文件或者协议的格式
3. opCode,反应了响应的操作已经创建了
4. 执行操作的 key
5. 使用的额外的 flag
6. 表明到期时间
7. body
8. 请求的 id。这个id将在响应中回显。
9. compare-and-check 的值
10. 如果有额外的使用，将返回 true

你如果想实现 Memcached 的其余部分协议,你只需要将 client.op(*op* 任何新的操作添加)转换为其中一个方法请求。我们需要两个更多的支持类,在下一个清单所示

Listing 14.2 Possible Memcached operation codes and response statuses

```
public class Status {
    public static final short NO_ERROR = 0x0000;
    public static final short KEY_NOT_FOUND = 0x0001;
    public static final short KEY_EXISTS = 0x0002;
    public static final short VALUE_TOO_LARGE = 0x0003;
    public static final short INVALID_ARGUMENTS = 0x0004;
    public static final short ITEM_NOT_STORED = 0x0005;
    public static final short INC_DEC_NON_NUM_VAL = 0x0006;
}
public class Opcode {
    public static final byte GET = 0x00;
    public static final byte SET = 0x01;
    public static final byte DELETE = 0x04;
}
```

一个 Opcode 告诉 Memcached 要执行哪些操作。每个操作都由一个字节表示。同样的,当 Memcached 响应一个请求,响应头中包含两个字节代表响应状态。状态和 Opcode 类表示这些 Memcached 的构造。这些操作码可以使用当你构建一个新的 MemcachedRequest 指定哪个行动应该由它引发的。

但现在可以集中精力在编码器上：

Listing 14.3 MemcachedRequestEncoder implementation

```
public class MemcachedRequestEncoder extends
        MessageToByteEncoder<MemcachedRequest> { //1
    @Override
    protected void encode(ChannelHandlerContext ctx, MemcachedRequest msg,
                          ByteBuf out) throws Exception {  //2
        byte[] key = msg.key().getBytes(CharsetUtil.UTF_8);
        byte[] body = msg.body().getBytes(CharsetUtil.UTF_8);
        //total size of the body = key size + content size + extras size   //3
        int bodySize = key.length + body.length + (msg.hasExtras() ? 8 : 0);

        //write magic byte  //4
        out.writeByte(msg.magic());
        //write opcode byte  //5
        out.writeByte(msg.opCode());
        //write key length (2 byte) //6
        out.writeShort(key.length); //key length is max 2 bytes i.e. a Java short  //7
        //write extras length (1 byte)
        int extraSize =  msg.hasExtras() ? 0x08 : 0x0;
        out.writeByte(extraSize);
        //byte is the data type, not currently implemented in Memcached but required /
/8
        out.writeByte(0);
        //next two bytes are reserved, not currently implemented but are required  //9
        out.writeShort(0);

        //write total body length ( 4 bytes - 32 bit int)  //10
        out.writeInt(bodySize);
        //write opaque ( 4 bytes)  -  a 32 bit int that is returned in the response //
11
        out.writeInt(msg.id());

        //write CAS ( 8 bytes)
        out.writeLong(msg.cas());   //24 byte header finishes with the CAS  //12

        if (msg.hasExtras()) {
            //write extras (flags and expiry, 4 bytes each) - 8 bytes total  //13
            out.writeInt(msg.flags());
            out.writeInt(msg.expires());
        }
        //write key   //14
        out.writeBytes(key);
        //write value  //15
        out.writeBytes(body);
    }
}
```

1. 该类是负责编码 MemachedRequest 为一系列字节

2. 转换的 key 和实际请求的 body 到字节数组
3. 计算 body 大小
4. 写幻数到 ByteBuf 字节
5. 写 opCode 作为字节
6. 写 key 长度z作为 short
7. 编写额外的长度作为字节
8. 写数据类型,这总是0,因为目前不是在 Memcached,但可用于使用 后来的版本
9. 为保留字节写为 short ,后面的 Memcached 版本可能使用
10. 写 body 的大小作为 long
11. 写 opaque 作为 int
12. 写 cas 作为 long。这个是头文件的最后部分，在 body 的开始
13. 编写额外的 flag 和到期时间为 int
14. 写 key
15. 这个请求完成后 写 body。

总结，编码器 使用 Netty 的 ByteBuf 处理请求，编码 MemcachedRequest 成一套正确排序的字节。详细步骤为：

- 写幻数字节。
- 写 opcode 字节。
- 写 key 长度(2字节)。
- 写额外的长度(1字节)。
- 写数据类型(1字节)。
- 为保留字节写 null 字节(2字节)。
- 写 body 长度(4字节- 32位整数)。
- 写 opaque(4个字节,一个32位整数在响应中返回)。
- 写 CAS(8个字节)。
- 写 额外的(flag 和 到期,4字节)= 8个字节
- 写 key
- 写 值

无论你放入什么到输出缓冲区(调用 ByteBuf) Netty 的将向服务器发送被写入请求。下一节将展示如何进行反向通过解码器工作。

实现 Memcached 解码器

将 MemcachedRequest 对象转为字节序列，Memcached 仅需将字节转到响应对象返回即可。

先见一个 POJO:

Listing 14.7 Implementation of a MemcachedResponse

```
public class MemcachedResponse {  //1
    private final byte magic;
    private final byte opCode;
    private byte dataType;
    private final short status;
    private final int id;
    private final long cas;
    private final int flags;
    private final int expires;
    private final String key;
    private final String data;

    public MemcachedResponse(byte magic, byte opCode,
            byte dataType,                             short status,
            int id, long cas,
            int flags, int expires, String key, String data) {
        this.magic = magic;
        this.opCode = opCode;
        this.dataType = dataType;
        this.status = status;
        this.id = id;
        this.cas = cas;
        this.flags = flags;
        this.expires = expires;
        this.key = key;
        this.data = data;
    }

    public byte magic() { //2
        return magic;
    }

    public byte opCode() { //3
        return opCode;
    }

    public byte dataType() { //4
        return dataType;
    }

    public short status() {  //5
        return status;
    }

    public int id() {  //6
        return id;
    }

    public long cas() {  //7
        return cas;
    }
```

```
    public int flags() {  //8
        return flags;
    }

    public int expires() { //9
        return expires;
    }

    public String key() {  //10
        return key;
    }

    public String data() {  //11
        return data;
    }
}
```

1. 该类,代表从 Memcached 服务器返回的响应
2. 幻数
3. opCode,这反映了创建操作的响应
4. 数据类型,这表明这个是基于二进制还是文本
5. 响应的状态,这表明如果请求是成功的
6. 惟一的 id
7. compare-and-set 值
8. 使用额外的 flag
9. 表示该值存储的一个有效期
10. 响应创建的 key
11. 实际数据

下面为 MemcachedResponseDecoder，使用了 ByteToMessageDecoder 基类，用于将 字节序列转为 MemcachedResponse

Listing 14.4 MemcachedResponseDecoder class

```
public class MemcachedResponseDecoder extends ByteToMessageDecoder {  //1
    private enum State {  //2
        Header,
        Body
    }

    private State state = State.Header;
    private int totalBodySize;
    private byte magic;
    private byte opCode;
    private short keyLength;
    private byte extraLength;
    private short status;
    private int id;
```

```
    private long cas;

    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf in,
                          List<Object> out) {
        switch (state) { //3
            case Header:
                if (in.readableBytes() < 24) {
                    return;//response header is 24  bytes  //4
                }
                magic = in.readByte();  //5
                opCode = in.readByte();
                keyLength = in.readShort();
                extraLength = in.readByte();
                in.skipBytes(1);
                status = in.readShort();
                totalBodySize = in.readInt();
                id = in.readInt(); //referred to in the protocol spec as opaque
                cas = in.readLong();

                state = State.Body;
            case Body:
                if (in.readableBytes() < totalBodySize) {
                    return; //until we have the entire payload return  //6
                }
                int flags = 0, expires = 0;
                int actualBodySize = totalBodySize;
                if (extraLength > 0) {  //7
                    flags = in.readInt();
                    actualBodySize -= 4;
                }
                if (extraLength > 4) {  //8
                    expires = in.readInt();
                    actualBodySize -= 4;
                }
                String key = "";
                if (keyLength > 0) {  //9
                    ByteBuf keyBytes = in.readBytes(keyLength);
                    key = keyBytes.toString(CharsetUtil.UTF_8);
                    actualBodySize -= keyLength;
                }
                ByteBuf body = in.readBytes(actualBodySize);  //10
                String data = body.toString(CharsetUtil.UTF_8);
                out.add(new MemcachedResponse(  //1
                        magic,
                        opCode,
                        status,
                        id,
                        cas,
                        flags,
                        expires,
                        key,
                        data
```

```
                ));

                state = State.Header;
        }

    }
}
```

1. 类负责创建的 MemcachedResponse 读取字节
2. 代表当前解析状态,这意味着我们需要解析的头或 body
3. 根据解析状态切换
4. 如果不是至少24个字节是可读的,它不可能读整个头部,所以返回这里,等待再通知一次数据准备阅读
5. 阅读所有头的字段
6. 检查是否足够的数据是可读用来读取完整的响应的 body。长度是从头读取
7. 检查如果有任何额外的 flag 用于读，如果是这样做
8. 检查如果响应包含一个 expire 字段，有就读它
9. 检查响应是否包含一个 key ,有就读它
10. 读实际的 body 的 payload
11. 从前面读取字段和数据构造一个新的 MemachedResponse

所以在实现发生了什么事?我们知道一个 Memcached 响应有24位头;我们不知道是否所有数据,响应将被包含在输入 ByteBuf ，当解码方法调用时。这是因为底层网络堆栈可能将数据分解成块。所以确保我们只解码当我们有足够的数据,这段代码检查是否可用可读的字节的数量至少是24。一旦我们有24个字节,我们可以确定整个消息有多大,因为这个信息包含在24位头。

当我们解码整个消息,我们创建一个 MemcachedResponse 并将其添加到输出列表。任何对象添加到该列表将被转发到下一个ChannelInboundHandler 在 ChannelPipeline,因此允许处理。

测试编解码器

编码器和解码器完成,但仍有一些缺失:测试。

没有测试你只看到如果编解码器工作对一些真正的服务器运行时,这并不是你应该是依靠什么。第十章所示,为一个自定义编写测试 ChannelHandler通常是通过 EmbeddedChannel。

所以这正是现在做测试我们定制的编解码器,其中包括一个编码器和解码器。让重新开始编码器。后面的清单显示了简单的编写单元测试。

Listing 14.5 MemcachedRequestEncoderTest class

public class MemcachedRequestEncoderTest {

```
@Test
public void testMemcachedRequestEncoder() {
    MemcachedRequest request = new MemcachedRequest(Opcode.SET, "key1", "value1"); //1

    EmbeddedChannel channel = new EmbeddedChannel(new MemcachedRequestEncoder());  //2
    channel.writeOutbound(request); //3

    ByteBuf encoded = (ByteBuf) channel.readOutbound();

    Assert.assertNotNull(encoded);  //4
    Assert.assertEquals(request.magic(), encoded.readUnsignedByte());  //5
    Assert.assertEquals(request.opCode(), encoded.readByte());  //6
    Assert.assertEquals(4, encoded.readShort());//7
    Assert.assertEquals((byte) 0x08, encoded.readByte()); //8
    Assert.assertEquals((byte) 0, encoded.readByte());//9
    Assert.assertEquals(0, encoded.readShort());//10
    Assert.assertEquals(4 + 6 + 8, encoded.readInt());//11
    Assert.assertEquals(request.id(), encoded.readInt());//12
    Assert.assertEquals(request.cas(), encoded.readLong());//13
    Assert.assertEquals(request.flags(), encoded.readInt()); //14
    Assert.assertEquals(request.expires(), encoded.readInt()); //15

    byte[] data = new byte[encoded.readableBytes()]; //16
    encoded.readBytes(data);
    Assert.assertArrayEquals((request.key() + request.body()).getBytes(CharsetUtil.UTF
_8), data);
    Assert.assertFalse(encoded.isReadable());  //17

    Assert.assertFalse(channel.finish());
    Assert.assertNull(channel.readInbound());
}
```

}

1. 新建 MemcachedRequest 用于编码为 ByteBuf
2. 新建 EmbeddedChannel 用于保持 MemcachedRequestEncoder 到测试
3. 写请求到 channel 并且判断是否产生了编码的消息
4. 检查 ByteBuf 是否 null
5. 判断 magic 是否正确写入 ByteBuf
6. 判断 opCode (SET) 是否写入正确
7. 检查 key 是否写入长度正确
8. 检查写入的请求是否额外包含
9. 检查数据类型是否写
10. 检查是否保留数据插入
11. 检查 body 的整体大小 计算方式是 key.length + body.length + extras
12. 检查是否正确写入 id
13. 检查是否正确写入 Compare and Swap (CAS)
14. 检查是否正确的 flag
15. 检查是否正确设置到期时间的
16. 检查 key 和 body 是否正确
17. 检查是否可读

Listing 14.6 MemcachedResponseDecoderTest class

```
public class MemcachedResponseDecoderTest {

    @Test
    public void testMemcachedResponseDecoder() {
        EmbeddedChannel channel = new EmbeddedChannel(new MemcachedResponseDecoder());
  //1

        byte magic = 1;
        byte opCode = Opcode.SET;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer(); //2
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(0);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);
```

```
        Assert.assertTrue(channel.writeInbound(buffer));  //3

        MemcachedResponse response = (MemcachedResponse) channel.readInbound();
        assertResponse(response, magic, opCode, Status.KEY_EXISTS, 0, 0, id, cas, key,
 body);//4
    }

    @Test
    public void testMemcachedResponseDecoderFragments() {
        EmbeddedChannel channel = new EmbeddedChannel(new MemcachedResponseDecoder());
 //5

        byte magic = 1;
        byte opCode = Opcode.SET;

        byte[] key = "Key1".getBytes(CharsetUtil.US_ASCII);
        byte[] body = "Value".getBytes(CharsetUtil.US_ASCII);
        int id = (int) System.currentTimeMillis();
        long cas = System.currentTimeMillis();

        ByteBuf buffer = Unpooled.buffer(); //6
        buffer.writeByte(magic);
        buffer.writeByte(opCode);
        buffer.writeShort(key.length);
        buffer.writeByte(0);
        buffer.writeByte(0);
        buffer.writeShort(Status.KEY_EXISTS);
        buffer.writeInt(body.length + key.length);
        buffer.writeInt(id);
        buffer.writeLong(cas);
        buffer.writeBytes(key);
        buffer.writeBytes(body);

        ByteBuf fragment1 = buffer.readBytes(8); //7
        ByteBuf fragment2 = buffer.readBytes(24);
        ByteBuf fragment3 = buffer;

        Assert.assertFalse(channel.writeInbound(fragment1));  //8
        Assert.assertFalse(channel.writeInbound(fragment2));  //9
        Assert.assertTrue(channel.writeInbound(fragment3));  //10

        MemcachedResponse response = (MemcachedResponse) channel.readInbound();
        assertResponse(response, magic, opCode, Status.KEY_EXISTS, 0, 0, id, cas, key,
 body);//11
    }

    private static void assertResponse(MemcachedResponse response, byte magic, byte op
Code, short status, int expires, int flags, int id, long cas, byte[] key, byte[] body)
 {
        Assert.assertEquals(magic, response.magic());
        Assert.assertArrayEquals(key, response.key().getBytes(CharsetUtil.US_ASCII));
        Assert.assertEquals(opCode, response.opCode());
```

```
        Assert.assertEquals(status, response.status());
        Assert.assertEquals(cas, response.cas());
        Assert.assertEquals(expires, response.expires());
        Assert.assertEquals(flags, response.flags());
        Assert.assertArrayEquals(body, response.data().getBytes(CharsetUtil.US_ASCII))
;
        Assert.assertEquals(id, response.id());
    }
}
```

1. 新建 EmbeddedChannel ，持有 MemcachedResponseDecoder 到测试
2. 创建一个新的 Buffer 并写入数据，与二进制协议的结构相匹配
3. 写缓冲区到 EmbeddedChannel 和检查是否一个新的MemcachedResponse 创建由声明返回值
4. 判断 MemcachedResponse 和预期的值
5. 创建一个新的 EmbeddedChannel 持有 MemcachedResponseDecoder 到测试
6. 创建一个新的 Buffer 和写入数据的二进制协议的结构相匹配
7. 缓冲分割成三个片段
8. 写的第一个片段 EmbeddedChannel 并检查,没有新的MemcachedResponse 创建,因为并不是所有的数据都是准备好了
9. 写第二个片段 EmbeddedChannel 和检查,没有新的MemcachedResponse 创建,因为并不是所有的数据都是准备好了
10. 写最后一段到 EmbeddedChannel 和检查新的 MemcachedResponse 是否创建，因为我们终于收到所有数据
11. 判断 MemcachedResponse 与预期的值

总结

阅读本章后,您应该能够创建自己的编解码器针对你最喜欢的协议。这包括写编码器和解码器,从字节转换为你的 POJO,反之亦然。这一章展示了如何使用一个协议规范实现和提取所需的信息。

它还向您展示了如何编写单元测试完成你的工作的编码器和解码器,确保一切工作如预期而不需要一个完整的 Memcached 服务器运行。这允许轻松集成测试到构建系统的中。

EventLoop 和线程模型

本章介绍

- 线程模型的总览
- EventLoop
- 并发
- 任务执行
- 任务调度

线程模型定义了应用或者框架如何执行你的代码，所以选择线程模型极其重要。Netty 提供了一个简单强大的线程模型来帮助我们简化代码。所有 ChannelHandler，包括业务逻辑，都保证由一个 Thread 同时执行特定的 Channel。这并不意味着Netty不能使用多线程，只是 Netty 限制每个Channel 都由一个 Thread 处理，这种设计适用于非阻塞 IO 操作。

读完本章就会深刻理解 Netty 的线程模型以及 Nett y团队为什么会选择这样的线程模型，这些信息可以让我们在使用 Netty 时让程序由最好的性能。此外，Netty 提供的线程模型还可以让我们编写整洁简单的代码，以保持代码的整洁性；我们还会学习 Netty 团队的经验，过去使用其他的线程模型，现在我们将使用 Netty 提供的更容易更强大的线程模型来开发。

本章假设如下：

- 你明白线程是什么以及如何使用，并有使用线程的工作经验。若不是这样，就请花些时间来了解清楚这些知识。推荐一本书：《Java Concurrency in Practice（Java 并发编程实战）》（Brian Goetz）。
- 你了解多线程应用程序及其设计，也包括如何保证线程安全和获取最佳性能。

线程模型的总览

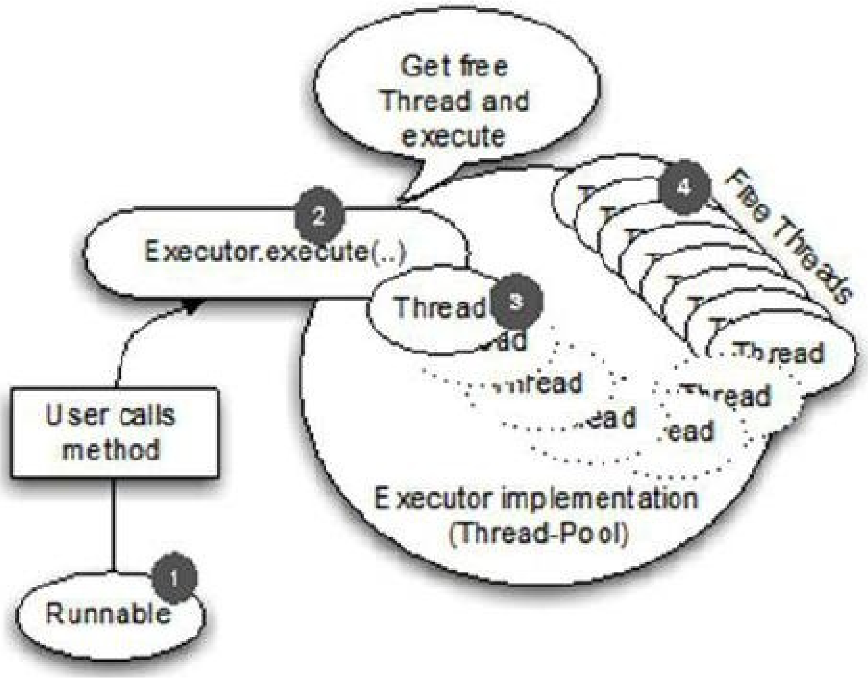
本节将简单介绍一般的线程模型，Netty 中如何使用指定的线程模型，以及Netty 过去不同的版本中使用的线程模型。你会更好的理解不同的线程模型的所有利弊。

一个线程模型指定代码执行,给开发人员如何执行他们代码的信息。这很重要,因为它允许开发人员事先知道如何保护他们的代码免受并发执行的副作用。若没有这个知识背景，即使是最好的开发人员都只能是碰运气,希望到最后都能这么幸运,但这几乎是不可能的。进入更多的细节之前,提供一个更好的理解主题的回顾这些天大多数应用程序做什么。

大多数现代应用程序使用多个线程调度工作,因此让应用程序使用所有可用的系统资源以有效的方式。这使得很多有意义,因为大部分硬件有不止一个甚至多个CPU核心。如果一切都只有一个 Thread 执行,不可能完全使用所提供的资源。为了解决这个问题,许多应用程序执行多个 Thread 的运行代码。在早期的 Java,这样做是通过简单地按需创建新 Thread 时,并行工作需要做。

但很快就发现,这不是完美的,因为创建 Thread 和回收会给他们带来的开销。在 Java 5 中,我们终于有了所谓的线程池,经常缓存 Thread,用来消除创建和回收 Thread 的开销。这些池由 Executor 接口提供。Java 5 提供了许多有用的实现,在其内部发生显著的变化,但思想都一脉相承的。创建 Thread 和重用他们提交一个任务时执行。这可以帮助创建和回收线程的开销降到最低。

下图显示使用一个线程池执行一个任务，提交一个任务后会使用线程池中空闲的线程来执行，完成任务后释放线程并将线程重新放回线程池：



1. Runnable 表示要执行的任务。这可能是任何东西,从一个数据库调用文件系统清理。

2. 之前 runnable 移交到线程池。
3. 闲置的线程被用来执行任务。当一个线程运行结束之后,它将回到闲置线程的列表新任务需要运行时被重用。
4. 线程执行任务

Figure 15.1 Executor execution logic

这个修复 Thread 创建和回收的开销,不需要每个新任务创建和销毁新的 Thread 。

但使用多个 Thread 提供了资源和管理成本,作为一个副作用,引入了太多的上下文切换。这种会随着运行的线程的数量和任务执行的数量的增加而恶化。尽管使用多个线程在开始时似乎不是一个问题,但一旦你把真正工作负载放在系统上，可以会遭受到重击。

除了这些技术的限制和问题,其他问题可能发生在相关的维护应用程序/框架在未来或在项目的生命周期里。有效地说,增加应用程序的复杂性取决于对比。当状态简单时,写一个多线程应用程序是一个辛苦的工作!你能解决这个问题吗?在实际的场景中需要多个 Thread 规模;这是一个事实。让我们看看 Netty 是解决这个问题。

EventLoop

事件循环所做的正如它的名字所说的。它运行在一个循环里,直到它的终止。这符合网络框架的设计,因为他们需要在一个循环为一个特定的连接运行事件。这不是 Netty 发明新的东西;其他框架和实现已经这样做了。

下面的清单显示了典型的 EventLoop 逻辑。请注意这是为了更好的说明这个想法而不是单单展示 Netty 实现本身。

Listing 14.1 Execute task in EventLoop

```
while (!terminated) {
    List<Runnable> readyEvents = blockUntilEventsReady(); //1
    for (Runnable ev: readyEvents) {
        ev.run(); //2
    }
}
```

1. 阻塞直到事件可以运行
2. 循环所有事件，并运行他们

在 Netty 中使用 EventLoop 接口代表事件循环，EventLoop 是从EventExecutor 和 ScheduledExecutorService 扩展而来，所以可以将任务直接交给 EventLoop 执行。类关系图如下：

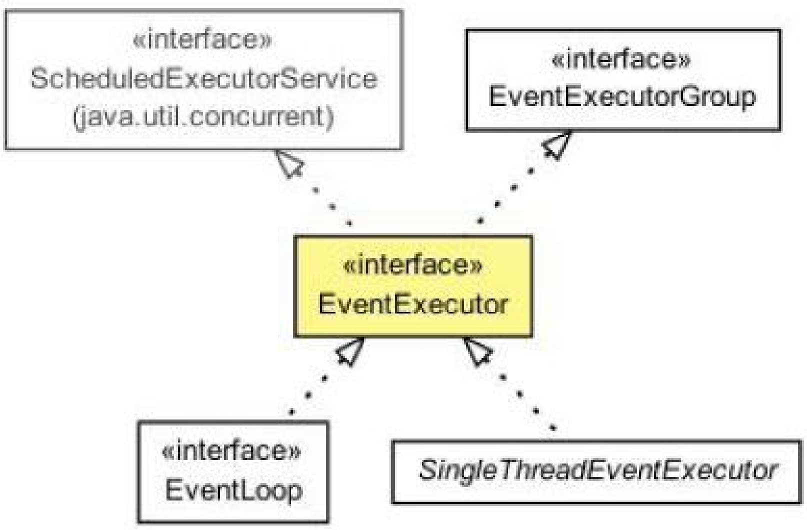


Figure 15.2 EventLoop class hierarchy

EventLoop 是完全由一个 Thread,从未改变。为了更合理利用资源,根据配置和可用的内核，Netty 可以使用多个 EventLoop。

事件/任务执行顺序

一个重要的细节关于事件和任务的执行顺序是,事件/任务执行顺序按照*FIFO(*先进先出*)*。这是必要的,因为否则事件不能按顺序处理,所处理的字节将不能保证正确的顺序。这将导致问题,所以这个不是所允许的设计。

Netty 4 中的 I/O 和事件处理

Netty 使用 I/O 事件,b被各种 I/O 操作运输本身所触发。 这些 I/O 操作，例如网络 API 的一部分，由Java 和底层操作系统提供。

一个区别在于,一些操作(或者事件)是由 Netty 的本身的传输实现触发的，一些是由用户自己。例如读事件通常是由传输本身在读取一些数据时触发。相比之下,写事件通常是由用户本身,例如,当调用 Channel.write(...)。

究竟需要做一次处理一个事件取决于事件的性质。经常会读网络栈的数据转移到您的应用程序。有时它会在另一个方向做同样的事情,例如,把数据从应用程序到网络堆栈(内核)发送到它的远端。但不限于这种类型的事务;重要的是,所使用的逻辑是通用的,灵活地处理各种各样的用例。

I/O 和事件处理的一个重要的事情在 Netty 4,是每一个 I/O 操作和事件总是由 EventLoop 本身处理,以及分配给 EventLoop 的 Thread。

我们应该注意,Netty 不总是使用我们描述的线程模型(通过 EventLoop 抽象)。在下一节中,你会了解 Netty 3 中使用的线程模型。这将帮助你理解为什么现在用新的线程模型以及为什么使用取代了 Netty 3 中仍然使用的旧模式。

Netty 3 中的 I/O 操作

在以前的版本中,线程模型是不同的。Netty 保证只将入站(以前称为 upstream)事件在执行 I/O Thread 执行 (I/O Thread 现在在 Netty 4 叫 EventLoop)。所有的出站(以前称为 downstream)事件被调用Thread 处理,这可能是 I/O Thread 也可以能是其他 Thread。 这听起来像一个好主意,但原来是容易出错,因为处理 ChannelHandler需要小心的出站事件同步,因为它没有保证只有一个线程运行在同一时间。这可能会发生如果你触发 downstream 事件同时在一个管道时;例如,您 调用 Channel.write(..) 在不同的线程。

除了需要负担同步 ChannelHandler，这个线程模型的另一个问题是你可能需要去掉一个入站事件作为一个出站事件的结果，例如 Channel.write(..) 操作导致异常。在这种情况下，exceptionCaught 必须生成并抛出去。乍看之下这不像是一个问题，但我们知道，exceptionCaught 由入站事件涉及，会让你知道问题出在哪里。问题是，事实上，你现在的情况是在调用 Thread 上执行，但 exceptionCaught 事件必须交给工作线程来执行，这样上下文切换是必须的。

相比之下,Netty 4 新线程模型根本没有这些问题,因为一切都在同一个EventLoop 在同一Thread 中 执行。这消除了需要同步ChannelHandler ,并且使它更容易为用户理解执行。

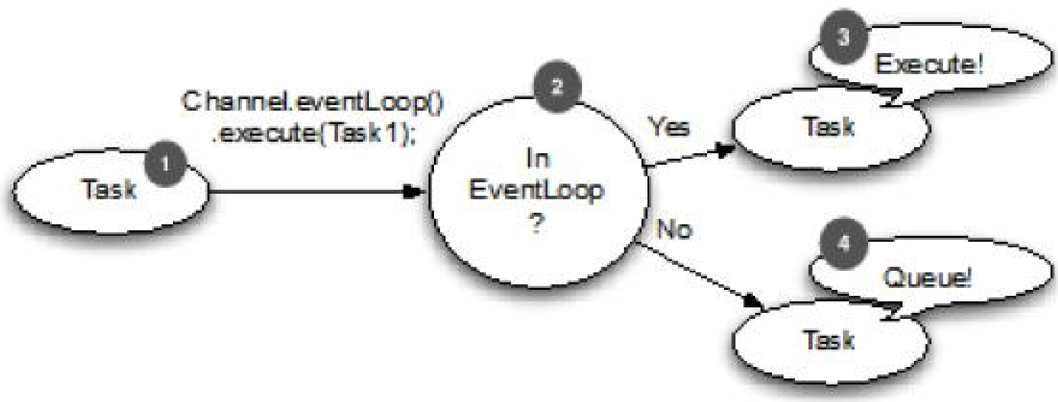
现在你知道 EventLoop 如何执行任务,它的时间来快速浏览下 Netty 的各种内部功能。

Netty 线程模型的内部

Netty 的内部实现使其线程模型表现优异，它会检查正在执行的 Thread 是否是已分配给实际 Channel (和 EventLoop)，在 Channel 的生命周期内，EventLoop 负责处理所有的事件。

如果 Thread 是相同的 EventLoop 中的一个，讨论的代码块被执行；如果线程不同，它安排一个任务并在一个内部队列后执行。通常是通过EventLoop 的 Channel 只执行一次下一个事件，这允许直接从任何线程与通道交互，同时还确保所有的 ChannelHandler 是线程安全，不需要担心并发访问问题。

下图显示在 EventLoop 中调度任务执行逻辑，这适合 Netty 的线程模型：



1. 应在 EventLoop 中执行的任务
2. 任务传递到执行方法后,执行检查来检测调用线程是否是与分配给 EventLoop 是一样的
3. 线程是一样的,说明你在 EventLoop 里，这意味着可以直接执行的任务
4. 线程与 EventLoop 分配的不一样。当 EventLoop 事件执行时，队列的任务再次执行一次

15.5 EventLoop execution logic/flow

设计是非常重要的，以确保不要把任何长时间运行的任务放在执行队列中，因为长时间运行的任务会阻止其他在相同线程上执行的任务。这多少会影响整个系统依赖于 EventLoop 实现用于特殊传输的实现。

传输之间的切换在你的代码库中可能没有任何改变，重要的是：切勿阻塞 I/O 线程。如果你必须做阻塞调用(或执行需要长时间才能完成的任务)，使用 EventExecutor。

下一节将讲解一个在应用程序中经常使用的功能，就是调度执行任务(定期执行)。Java对这个需求提供了解决方案，但 Netty 提供了几个更好的方案

调度任务执行

每隔一段时间需要调度任务执行，也许你想注册一个任务在客户端完成连接5分钟后执行，一个常见的用例是发送一个消息“你还活着？”到远端通，如果远端没有反应，则可以关闭通道(连接)和释放资源。

本节介绍使用强大的 EventLoop 实现任务调度，还会简单介绍 Java API的任务调度，以方便和 Netty 比较加深理解。

使用普通的 **Java API** 调度任务

在 Java 中使用 JDK 提供的 ScheduledExecutorService 实现任务调度。使用 Executors 提供的静态方法创建 ScheduledExecutorService，有如下方法

Table 15.1 java.util.concurrent.Executors-Static methods to create a ScheduledExecutorService

方法	描述	
newScheduledThreadPool(int corePoolSize) newScheduledThreadPool(int corePoolSize,ThreadFactorythreadFactory)		创建一个新的

ScheduledThreadExecutorService 用于调度命令来延迟或者周期性的执行。 corePoolSize 用于计算线程的数量 newSingleThreadScheduledExecutor() newSingleThreadScheduledExecutor(ThreadFact orythreadFactory) | 新建一个 ScheduledThreadExecutorService 可以用于调度命令来延迟或者周期性的执行。它将使用一个线程来执行调度的任务

下面的 ScheduledExecutorService 调度任务 60 执行一次

Listing 15.4 Schedule task with a ScheduledExecutorService

```
ScheduledExecutorService executor = Executors
        .newScheduledThreadPool(10); //1

ScheduledFuture<?> future = executor.schedule(
        new Runnable() { //2
            @Override
            public void run() {
                System.out.println("Now it is 60 seconds later");  //3
            }
        }, 60, TimeUnit.SECONDS);  //4
// do something
//

executor.shutdown();  //5
```

1. 新建 ScheduledExecutorService 使用10个线程
2. 新建 runnable 调度执行
3. 稍后运行
4. 调度任务60秒后执行
5. 关闭 ScheduledExecutorService 来释放任务完成的资源

使用 EventLoop 调度任务

使用 ScheduledExecutorService 工作的很好，但是有局限性，比如在一个额外的线程中执行任务。如果需要执行很多任务，资源使用就会很严重；对于像 Netty 这样的高性能的网络框架来说，严重的资源使用是不能接受的。Netty 对这个问题提供了很好的方法。

Netty 允许使用 EventLoop 调度任务分配到通道，如下面代码：

Listing 15.5 Schedule task with EventLoop

```
Channel ch = null; // Get reference to channel
ScheduledFuture<?> future = ch.eventLoop().schedule(
        new Runnable() {
            @Override
            public void run() {
                System.out.println("Now its 60 seconds later");
            }
        }, 60, TimeUnit.SECONDS);
```

1. 新建 runnable 用于执行调度
2. 稍后执行
3. 调度任务60秒后运行

如果想任务每隔多少秒执行一次，看下面代码：

Listing 15.6 Schedule a fixed task with the EventLoop

```
Channel ch = null; // Get reference to channel
ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(
        new Runnable() {
            @Override
            public void run() {
                System.out.println("Run every 60 seconds");
            }
        }, 60, 60, TimeUnit.SECONDS);
```

1. 新建 runnable 用于执行调度
2. 将运行直到 ScheduledFuture 被取消
3. 调度任务60秒运行

取消操作，可以使用 ScheduledFuture 返回每个异步操作。 ScheduledFuture 提供一个方法用于取消一个调度了的任务或者检查它的状态。一个简单的取消操作如下：

```
ScheduledFuture<?> future = ch.eventLoop()
.scheduleAtFixedRate(..); //1
// Some other code that runs...
future.cancel(false); //2
```

1. 调度任务并获取返回的 ScheduledFuture
2. 取消任务，阻止它再次运行

调度的内部实现

Netty 内部实现其实是基于George Varghese 提出的 “Hashed and hierarchical timing wheels: Data structures to efficiently implement timer facility(散列和分层定时轮：数据结构有效实现定时器)”。这种实现只保证一个近似执行，也就是说任务的执行可能不是100%准确；在实践中，这已经被证明是一个可容忍的限制，不影响多数应用程序。所以，定时执行任务不可能100%准确的按时执行。

为了更好的理解它是如何工作，我们可以这样认为：

- 在指定的延迟时间后调度任务；
- 任务被插入到 EventLoop 的 Schedule-Task-Queue(调度任务队列)；
- 如果任务需要马上执行，EventLoop 检查每个运行；
- 如果有一个任务要执行，EventLoop 将立刻执行它，并从队列中删除；
- EventLoop 等待下一次运行，从第4步开始一遍又一遍的重复。

因为这样的实现计划执行不可能100%正确，对于多数用例不可能100%准备的执行计划任务；在 Netty 中，这样的工作几乎没有资源开销。

但是如果需要更准确的执行呢？很容易，你需要使用ScheduledExecutorService 的另一个实现，这不是 Netty 的内容。记住，如果不遵循 Netty 的线程模型协议，你将需要自己同步并发访问。

I/O EventLoop/Thread 分配细节

Netty 的使用一个包含 EventLoop 的 EventLoopGroup 为 Channel 的 I/O 和事件服务。EventLoop 创建并分配方式不同基于传输的实现。异步实现使用只有少数 EventLoop(和 Threads)共享于 Channel 之间 。这允许最小线程数服务多个 Channel,不需要为他们每个人都有一个专门的 Thread。

图15.7显示了如何使用 EventLoopGroup。

.jpg)

1. 所有的 EventLoop 由 EventLoopGroup 分配。这里它将使用三个EventLoop 实例
2. 这个 EventLoop 处理所有分配给它管道的事件和任务。每个EventLoop 绑定到一个 Thread
3. 管道绑定到 EventLoop,所以所有操作总是被同一个线程在 Channel 的生命周期执行。一个管道属于一个连接

Figure 15.7 Thread allocation for nonblocking transports (such as NIO and AIO)

如图所述，使用有 3个 EventLoop （每个都有一个 Thread ） EventLoopGroup 。EventLoop （同时也是 Thread ）直接当 EventLoopGroup 创建时分配。这样保证资源是可以使用的

这三个 EventLoop 实例将会分配给每个新创建的 Channel。这是通过EventLoopGroup 实现,管理 EventLoop 实例。实际实现会照顾所有EventLoop 实例上均匀的创建 Channel (同样是不同的 Thread)。

一旦 Channel 是分配给一个 EventLoop,它将使用这个 EventLoop 在它的生命周期里和同样的线程。你可以,也应该,依靠这个,因为它可以确保你不需要担心同步(包括线程安全、可见性和同步)在你 ChannelHandler实现。

但是这也会影响使用 ThreadLocal,例如,经常使用的应用程序。因为一个EventLoop 通常影响多个 Channel,ThreadLocal 将相同的 Channel 分配给 EventLoop。因此,它适合状态跟踪等等。它仍然可以用于共享重或昂贵的对象之间的 Channel ,不再需要保持状态,因此它可以用于每个事件,而不需要依赖于先前 ThreadLocal 的状态。

EventLoop 和 Channel

我们应该注意*,*在 *Netty 4 , Channel* 可能从 *EventLoop* 注销稍后又从不同 *EventLoop* 注册。这个功能是不赞成*,*因为它在实践中没有很好的工作

语义跟其他传输略有不同,如 OIO(Old Blocking I/O)运输,可以看到如图14.8所示。

.jpg)

1. 所有 EventLoop 从 EventLoopGroup 分配。每个新的 channel 将会获得新的 EventLoop
2. EventLoop 分配给 channel 用于执行所有事件和任务
3. Channel 绑定到 EventLoop。一个 channel 属于一个连接

Figure 15.8 Thread allocation of blocking transports (such as OIO)

你可能会注意到这里,一个 EventLoop (也是一个 Thread)创建每个 Channel。你可能被用来从开发网络应用程序是基于常规阻塞I/O在使用java.io.* 包。但即使语义变化在这种情况下,有一件事仍然是相同的:每个 I/O 通道将由一次只有一个线程来处理,这是一个线程增强 Channel 的 EventLoop。可以依靠这个硬性的规则,使 Netty 的框架很容易与其他网络框架进行比较。

总结

在这一章里,你知道 Netty 使用哪个线程模型。你学会了使用线程模型的优缺点以及当使用 Netty 它们如何简化你的生活。

除了学习的内部运作,您获得了洞察力，知道如何可以执行自己的任务在 EventLoop(I/O Thread) 和 Netty 一样。你学会了如何在一大堆任务中安排任务。您还了解了如何验证一个任务是否执行以及如何取消它。

你现在知道 Netty 使用的各个先前版本的线程模型，你获得了更多的背景信息知道为什么新线程模型是更强大的。

你对 Netty 的线程模型有了深入了解,从而帮助您最大限度地提高您的应用程序性能,同时最小化所需的代码。关于线程池和并发访问的更多信息,请参阅 Java Concurrency in Practice（Brian Goetz）。他的书将会给你一个更深层次的了解,即使是最复杂的应用程序必须处理多线程的用例场景。